ADVENTURES IN RAY TRACING

Photo-Realistic Images on Your PC

Explore the Fascinating World of Computer Graphics

DISK INCLUDES:

POLYRAY-Creates Spectacular Graphics

POVCAD-3-D Modeler Creates Complex Objects for Photo-Realistic Images & Much More!

que®

Steve Anger

Robert A. Mickelsen

Heinz Schuller

Dan Farmer

Dan Farmer

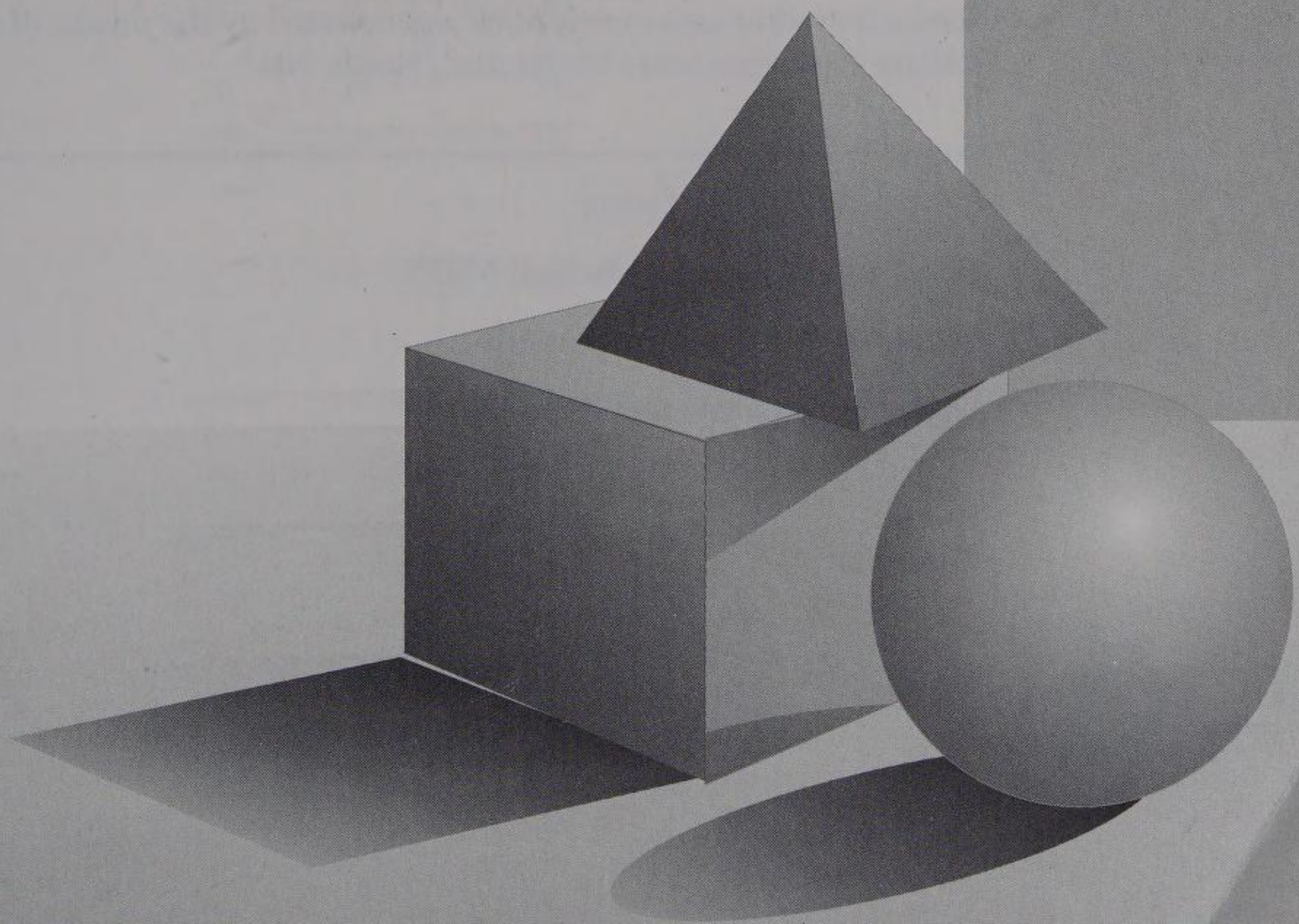
Dan Farmer

This book teaches you the ins and outs of ray tracing with Polyray so that you can create detailed, photo-realistic graphics. Use light and shadow, reflections, mirrored and wood grained surfaces--even three-dimensional depth-- to control every detail of your final image.

On the disk you'll find all you need to render brilliant images without spending hours writing code.

Adventures in Ray Tracing

Alfonso Hermida



que

Adventures in Ray Tracing

©1993 by Que® Corporation

All rights reserved. Printed in the United States of America. No part of this book may be used or reproduced in any form or by any means, or stored in a database or retrieval system, without prior written permission of the publisher except in the case of brief quotations embodied in critical articles and reviews. Making copies of any part of this book for any purpose other than your own personal use is a violation of United States copyright laws. For information, address Que Corporation, 201 W. 103rd St., Indianapolis, IN, 46290.

Library of Congress Catalog No.: 93-86073

ISBN: 1-56529-555-2

This book is sold *as is*, without warranty of any kind, either expressed or implied, respecting the contents of this book, including but not limited to implied warranties for the book's quality, performance, merchantability, or fitness for any particular purpose. Neither Que Corporation nor its dealers or distributors shall be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this book.

96 95 94 93 8 7 6 5 4 3 2 1

Interpretation of the printing code: The rightmost double-digit number is the year of the book's printing; the rightmost single-digit number, the number of the book's printing. For example, a printing code of 93-1 shows that the first printing of the book occurred in 1993.

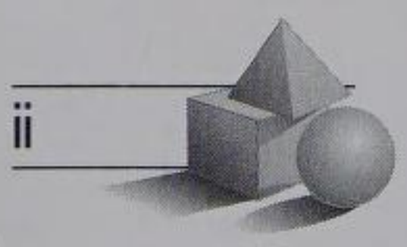
Screen reproductions in this book were created by the means of the program Collage Plus from Inner Media, Inc., Hollis, NH.

Publisher: David P. Ewing

Director of Publishing: Michael Miller

Managing Editor: Corinne Walls

Marketing Manager: Ray Robinson



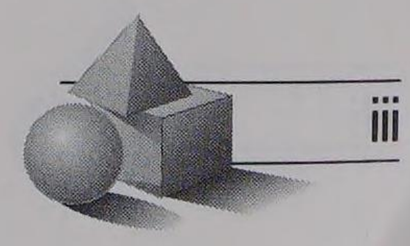
Dedication

To my wife, Linda, and my parents Alfonso and Betsy.

About the Author

Alfonso Hermida

Alfonso Hermida has a B.S. and an M.S. in mechanical engineering from the University of Puerto Rico, and an M.S. in mechanical engineering/space systems from The George Washington University in Washington, D.C. He currently works with NASA/Goddard Space Flight Center in Maryland as an aerospace engineer. He is also an adjunct professor of the Mathematics Department at Capitol College in Laurel, Maryland, where he teaches ordinary differential equations, CAD, and thermodynamics. He has worked as a software consultant and in his free time enjoys writing his own software programs. His main interests include structural analysis, dynamics, computer graphics, and animation.



Credits

Publishing Manager
Joseph B. Wikert

Product Development Specialist
Bryan Gambrel

Production Editors
Phil Kitchel
Lori Cates

Copy Editor
Linda Seifert

Technical Editor
Dan Richardson

Cover Designer
Dan Armstrong

Cover Illustration
Mike Miller, Headspin Studios

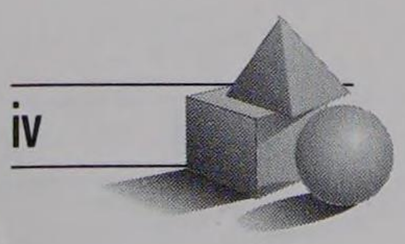
Book Designer
Amy Peppler-Adams

Illustrator
Katherine Hanley

Production Team
Angela Bannan
Danielle Bird
Charlotte Clapp
Teresa Forrester
Joelynn Gifford
Bob LaRoche
Tim Montgomery
Caroline Roop
Dennis Sheehan
Tina Trettin
Sue VandeWalle
Mary Beth Wakefield
Donna Winter
Lillian Yates

Indexer
Johnna VanHoose

Composed in Stone Serif and MCPdigital by Prentice Hall Computer Publishing



Acknowledgments

I have to thank a lot of people who were responsible in one way or another for this book:

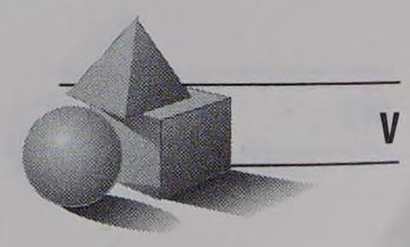
- Joe Wikert, for giving me the opportunity; the Que team, for cleaning up my typing, especially Phil Kitchel and Lori Cates; and Kathy Hanley, for her great work with my sketches. Thanks also to my technical editor, Dan Richardson, who was very enthusiastic about cleaning up after me.
- Alexander Enzmann, for doing such a great job writing Polyray.
- Bob Berry, the author of CompuShow, for letting us use his program.
- A special thanks to Dan Farmer, for telling me about CompuServe's GraphDev forum and for his feedback on POVCAD. He was kind enough to upload POVCAD to CompuServe.

  Finally, I want to take the opportunity to mention several people who have helped with POVCAD:

- Albert Waltner, Peter Jack, and Curtis Olson, for the initial Beta testing and all the helpful comments.
- Phil Long, Jerry Thomaston, Amanda Osborne, and James P. Hawkins, who continue to Beta test.
- David Standring, for Beta testing version 2.0b and checking the book manuscripts.

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Que cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.



Overview

Introduction 1
1 Introduction to Ray Tracing 5
2 Your First Ray-Traced Image 29
3 Using Polyray 47
4 Light, Colors, and Textures 105
5 Constructive Solid Geometry 163
6 Additional Features 181
7 Using a 3-D Modeler (POVCAD) 201
8 Animation 265
Where Can I Find Ray Tracing Software? 307
Index 309

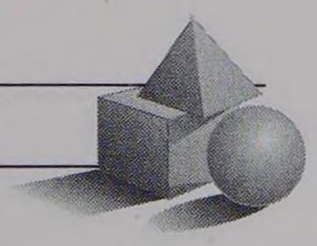
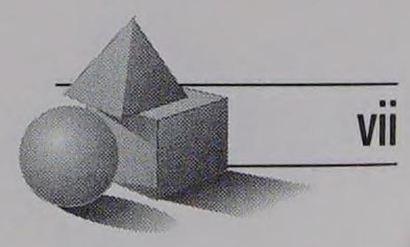


Table of Contents

Introduction 1
Who Should Read This Book? . . . 2
What This Book Is About . . . 3
Included Software . . . 3
Shareware Information . . . 4
Equipment Requirements . . . 4
Software Installation . . . 4

1 Introduction to Ray Tracing 5
What Is Ray Tracing? . . . 5
General Concepts . . . 8
3-D Coordinate Systems . . . 8
Viewpoint . . . 14
Light Sources . . . 14
3-D Geometry . . . 17
Lathe Surfaces . . . 19
Sweep/Extruded Surfaces . . . 20
Smooth Surfaces . . . 20
Building Complex Objects . . . 22
Constructive Solid Geometry . . . 23
Boolean Operations . . . 24
Adding Photo-Realism . . . 26
Summary . . . 28

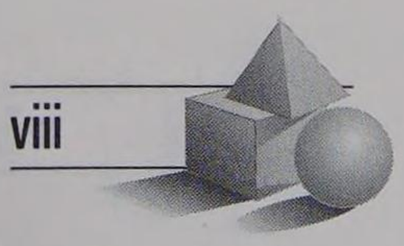
2 Your First Ray-Traced Image 29
The Polyray Ray Tracer . . . 30
Using Polyray . . . 30
Running POLYRAY.EXE . . . 30
Command-Line Parameters . . . 31
Creating a Scene File . . . 34
Using a Text Editor . . . 35
Syntax Considerations . . . 35
A Detailed Look at a Scene File . . . 35
Header Section . . . 36
Viewpoint Definition Section . . . 37



Light Source Definition 38
Object Definition 38
Textures 41
Ray Tracing a Scene File 42
Creating a Batch File 42
Ray Tracing Your First Scene 43
Viewing a Ray-Traced Image 43
Summary 45

3 Using Polyray **47**

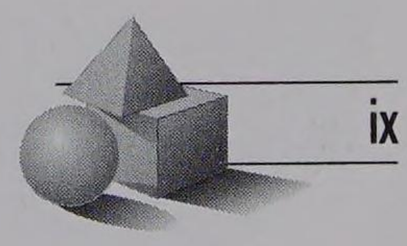
Background 47
3-D Coordinate System 47
Vectors 48
Transformations 48
Viewpoint 50
Naming Definitions 52
Creating Objects 53
Creating Simple Objects 53
Box 54
Plane 56
Sphere 59
Disc 65
Parabola 68
Cone 70
Cylinder 70
Ellipsoid 72
Torus 72
Modeling Surfaces 76
Smooth Triangles 76
Polygons 86
Lathe 86
Sweep/ Extruded 91
Bézier Patches 94
Miscellaneous Objects 98
Blobs 98
Height Fields 101
Functions 103
Summary 104



4 Light, Colors, and Textures 105
Light Sources 106
Positional Light Sources 106
Directional Light Sources 106
Spot Light Sources 109
Colors 113
Surface 116
Background 117
Global Haze 118
Textures 121
Texture Surface Parameters 121
Texture Types 131
Image Mapping Textures 158
Summary 162

5 Constructive Solid Geometry 163
Boolean Operations 163
Union 164
Difference 168
Intersection 171
Clipping 173
Bounding Box 175
Inverse 178
Summary 180

6 Additional Features 181
Shading Flags 182
Rendering Method 183
Scan Conversion 184
Wireframe 184
Mathematical Expressions 185
Floating Point Expressions 185
Vectors 188
Arrays 194
Conditional Expressions 195
Summary 198

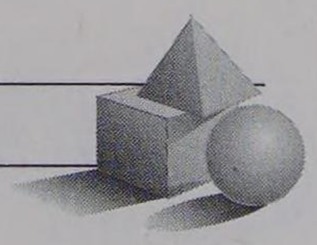


7 Using a 3-D Modeler (POVCAD) 201
Introduction 202
Running POVCAD 203
Creating a Scene File (Tutorial #1) 207
Taking a Closer Look at the TUTOR1.CAD Scene 210
The Polyray Scene File 212
Using POVCAD Primitives (Tutorial #2) 214
Creating Multiple Copies of Objects (Tutorial #3) 220
Using CSG Operations (Tutorial #4) 229
Sweep and Extrude Objects (Tutorial #5) 240
Creating Bézier Patches (Tutorial #6) 249
Using the Polygon Decomposition Technique (Tutorial #7) . . . 255
Summary 263

8 Animation 265
The Basics of Animation 265
Transformation Operations 266
Viewpoint Motion (Flyby) 269
Object Motion 271
Virtual Animation 272
Texture Variation 273
Polyray's Animation Features 274
Frame Generation 274
Conditional Processing 276
Static Definitions 278
Arrays 279
Creating the Animation File 281
Resolution Issues 282
Animation Examples 282
Flyby 282
Texture Animation 287
Virtual Animation 298
Summary 304

Where Can I Find Ray Tracing Software? 307

Index 309

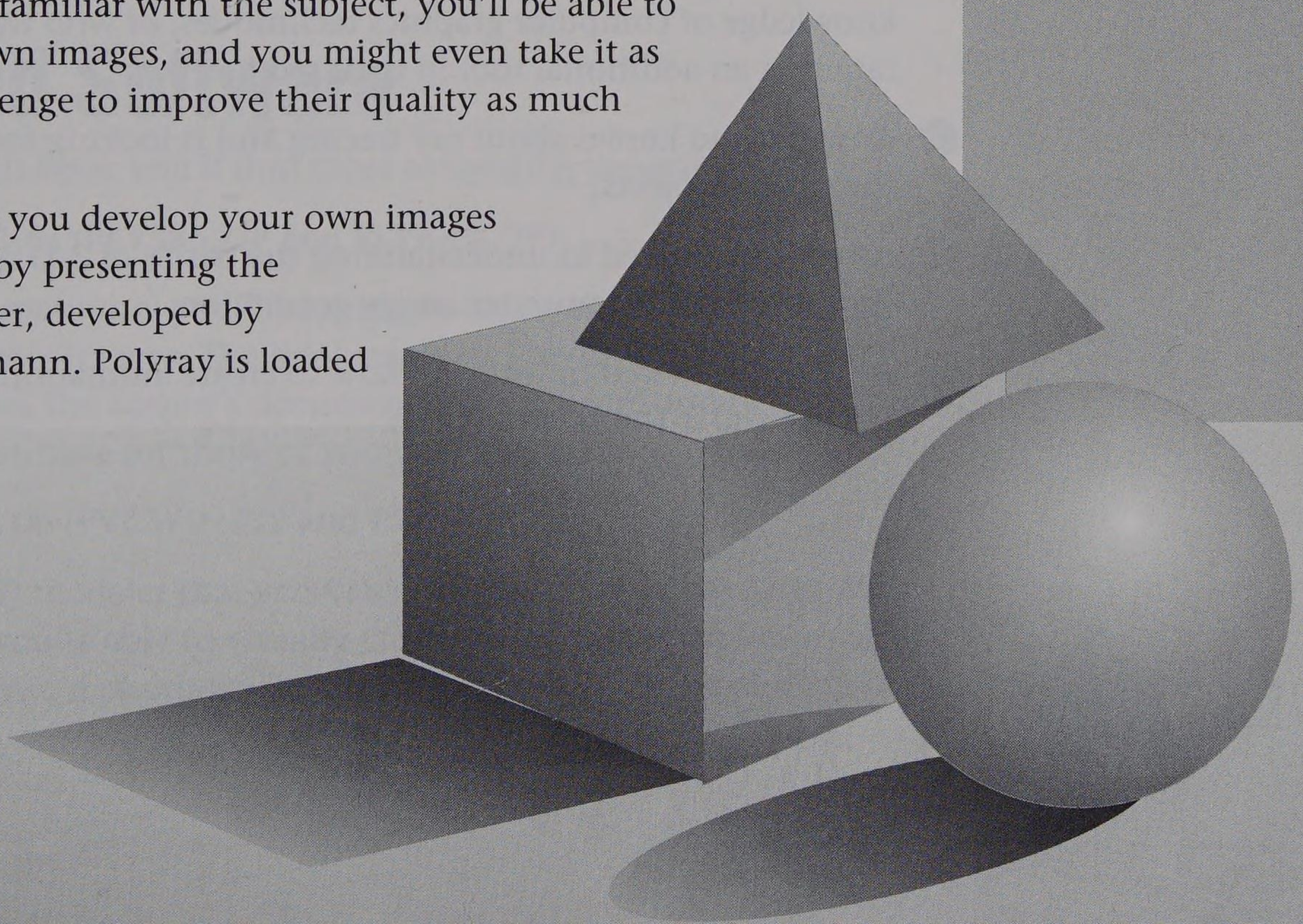


Introduction

Welcome to the exciting world of ray tracing! A little while ago, when personal computers starting coming out, computer image generation was only a dream. Computer graphics programs were usually power-hungry applications requiring lots of memory and speed. Now there are the 386 and 486 machines, and it's been worth the wait (at the time of this writing, Pentium chip-based computers are starting to come out—they promise to take ray tracing to a new level). Computer image generation is everywhere, from advertising and movies to games and art. The incredible growth in the popularity of computer-generated art has developed into a quest for the ultimate goal: a perfect photo-realistic image.

As you become familiar with the subject, you'll be able to develop your own images, and you might even take it as a personal challenge to improve their quality as much as possible.

This book helps you develop your own images and animation by presenting the Polyray ray tracer, developed by Alexander Enzmann. Polyray is loaded



with features, which this book presents while teaching you the basics of ray tracing, 3-D modeling, and animation. You'll have the chance to use the included sample files and modify them to suit your own taste. Eventually, you'll be able to start from scratch and convert your own thoughts into computer images.

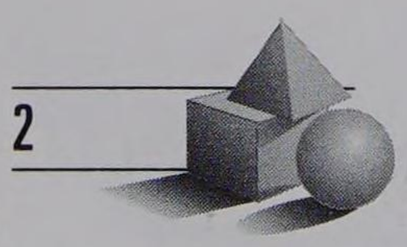
If you have already taken a quick glance over the book, you must be interested in the topic, and if after looking through it you can't figure out what's going on, then this book is for you. If you are already familiar with the topic, this book can help you increase your knowledge.

In any case, it's all about having fun and learning something useful at the same time.

Who Should Read This Book?

This book is for everyone interested in computer graphics, but it is specifically for

- Anyone who wants to learn how to create photo-realistic computer images. This group includes people who have never been exposed to the subject as well as people who have read about it but want to take a hands-on approach.
- Anyone who wants to learn what ray tracing is and what the basic concepts are. This group includes people who wish to increase their knowledge of computer graphics techniques, or who want to incorporate it as an additional tool in their work.
- Anyone who knows about ray tracing and is looking for new ideas for their own projects.
- Anyone interested in understanding the basics of 3-D modeling and how it relates to computer image generation.
- Anyone interested in learning how to create animations using ray tracing and 3-D modeling tools.



What This Book Is About

Adventures in Ray Tracing has many goals, but it is specifically intended to

- Help you learn how to create photo-realistic images in your computer by means of ray tracing (and the use of Polyray, the ray tracing program included with the book).
- Show you why a 3-D modeler is an important visualization tool for computer image generation, and show you how to use the one included with the book (POVCAD—both DOS and Windows versions are included).
- Show you how to combine the first two purposes to create your own animation sequences or movies.

This book takes you through all the necessary steps, starting with the basics of ray tracing: creating simple images and using the built-in objects available. Then you take a look at more complex objects and learn how to literally "cut and paste" pieces of objects together to form new objects. You are then introduced to POVCAD, the 3-D modeler included with the book, and shown how to generate scenes for the ray-tracing program. Finally, you develop animations using all the techniques you acquire working through the book.

By the end of the book, you'll have a good working knowledge of ray tracing, 3-D modeling, and animation.

Included Software

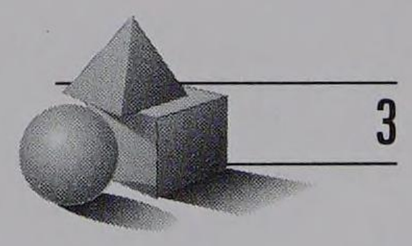
Bundled with this book you'll find three interesting programs:

1. POLYRAY 1.6a (PLY386.ZIP and PLYEXE.ZIP)

   This is the main tool you'll be using to create those awesome computer graphics images. The book explains Polyray's features, and the disk includes the author's documentation files and additional sample files. A must-have for those of you who like to modify code!

2. POVCAD 2.0b (PVCWIN.ZIP and PVCDOS.ZIP)

   This is a 3-D modeler that generates Polyray-compatible data. With POVCAD, you're able to visually create a scene that will eventually be converted into a photo-realistic image. Included on the disk are both



DOS and Windows versions. The Windows version is more advanced than the DOS version, and is presented in greater detail.

3. COMPUSHOW 8.61a (CSHOW.ZIP)

   This is an excellent utility program you can use as an image file viewer.

Needless to say, all the examples done in the book are on the disk.

Shareware Information

All the programs included with the disk are copyrighted by their authors and are distributed based on the shareware concept. If you wish to continue using these programs, you should register them. All of them require very modest fees, ranging from $15 to $35. For details, check the documentation available for each program.

Equipment Requirements

POLYRAY

There are two versions distributed with the disk (for use on computers with or without co-processors). At a minimum, both require a 386 microprocessor and 2 Megabytes of RAM. A VGA graphics card is also required (Super VGA is recommended).

POVCAD

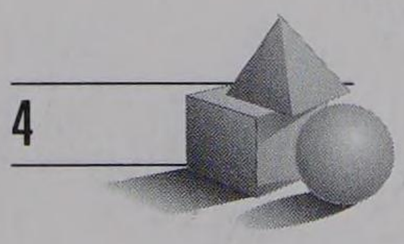
The Windows version requires Windows 3.1. A co-processor is highly recommended. The DOS version requires a VGA graphics card and a mouse.

COMPUSHOW

Requires a graphics card.

Software Installation

Check the READ.ME file included in the disk for the latest information on how to install the programs.



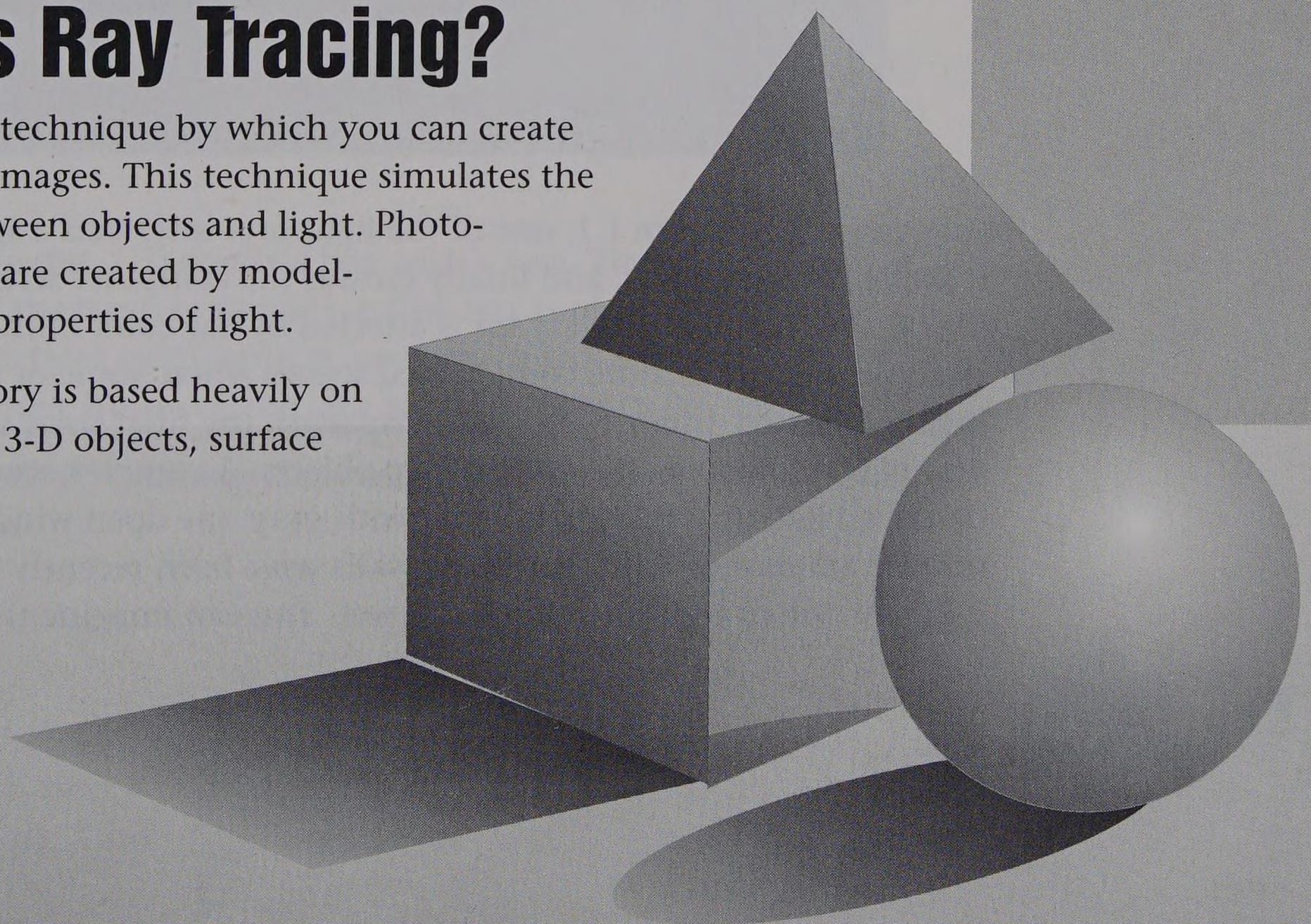
Introduction to Ray Tracing

In this chapter, you learn what ray tracing is and what components make up a ray tracing model. First, you learn how to define a 3-D scene and how to "look" at it by positioning an observer. Next, you place 3-D objects in the scene. Finally, you learn how to add photo-realism to the final image.

What Is Ray Tracing?

Ray tracing is a technique by which you can create photo-realistic images. This technique simulates the interaction between objects and light. Photo-realistic images are created by modeling the optical properties of light.

Ray tracing theory is based heavily on modeling light, 3-D objects, surface

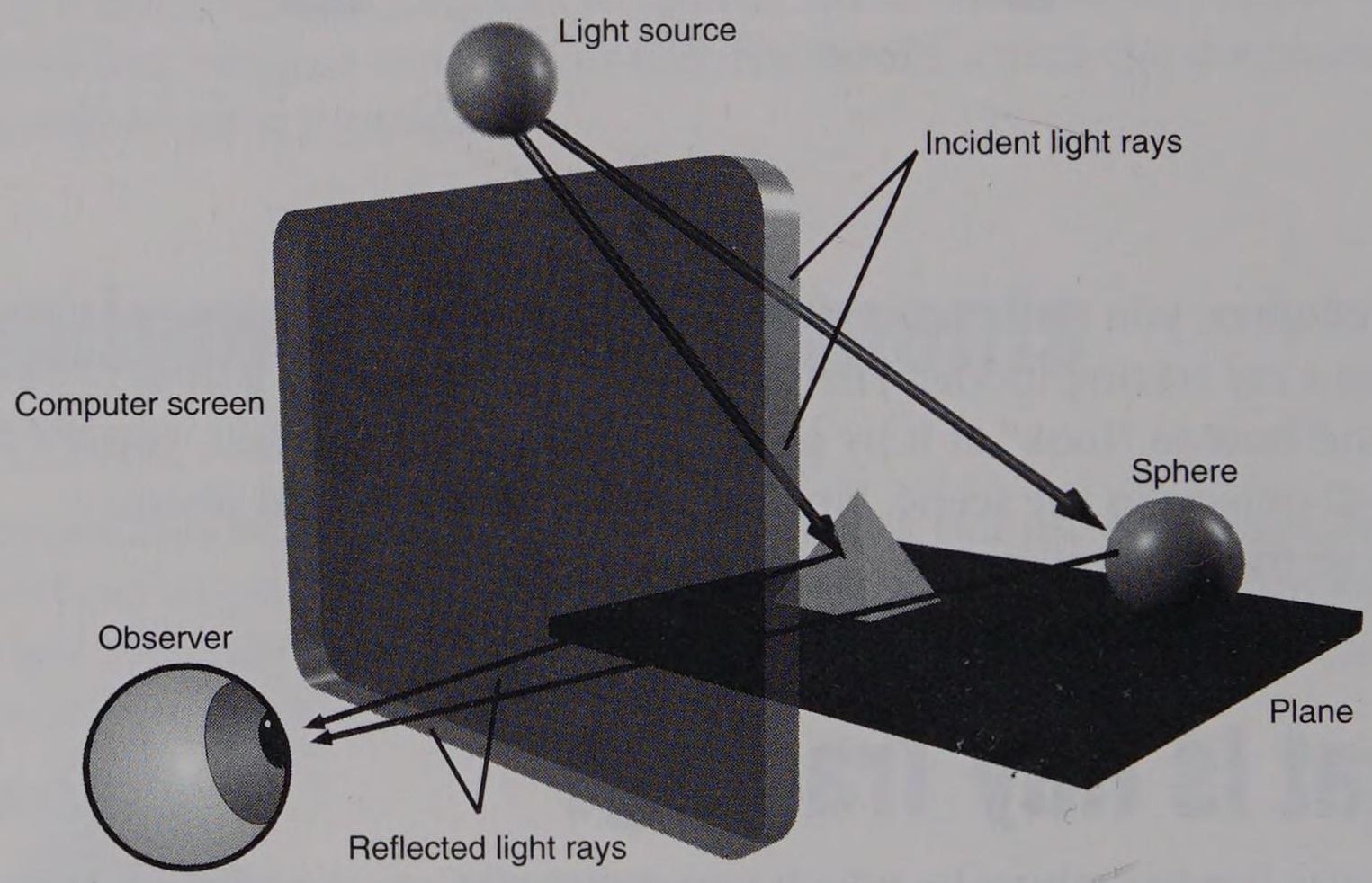


finishes, textures, and colors mathematically. Most ray tracing programs make most of the math clear to the user by using their own specialized and simplified language. Everything from light-object interaction to ripples on a surface is explained mathematically. Some of these mathematical expressions are not perfect, but they are good enough to simulate the "real thing."

In a ray tracing scene, you position objects as you want them to appear in the final image. The surface and color characteristics of each object are defined. Finally, light sources are located in specific places to create highlights and shadows, and an observer is positioned. This observer is your point of view. Next, the ray tracing program creates a mathematical model of the objects, the light, and the observer, then it computes the color for each pixel on-screen based on what the observer should be seeing (see fig. 1.1).

Figure 1.1

A representation of a ray tracing scene.



For example, in figure 1.1, one of the light rays intersects a sphere, is reflected by the sphere, and finally crosses the computer screen. The color of the pixel where the light ray hit is affected by the color of the sphere and the light source. In more complicated scenes where some of the objects are highly reflective (mirror-like) or transparent, the final color is a combination of the colors of any surrounding objects. A complex scene can be described like this: Imagine a room with only one open window and an infinite amount of walls. All of the walls have been recently painted with different colors and the paint is still wet. You can imagine that the light is a

highly elastic ball that was thrown inside the room. The ball keeps bouncing off the walls and hitting other walls until it goes through the window. Every time the ball bounces, it gets some paint from the wall. The color of the ball will be a combination of the colors of the walls it touched.

Here are two examples of ray-traced images. Figure 1.2 displays the top of a desk. The desk has a wood finish. On top of it is a yellow sphere that appears to be metallic, a lamp that has a very polished finish, and a frame with a picture in it. The picture is a photo of one of NASA's space shuttles. Can you determine where the light source is located? (Hint: Check the shadows.)

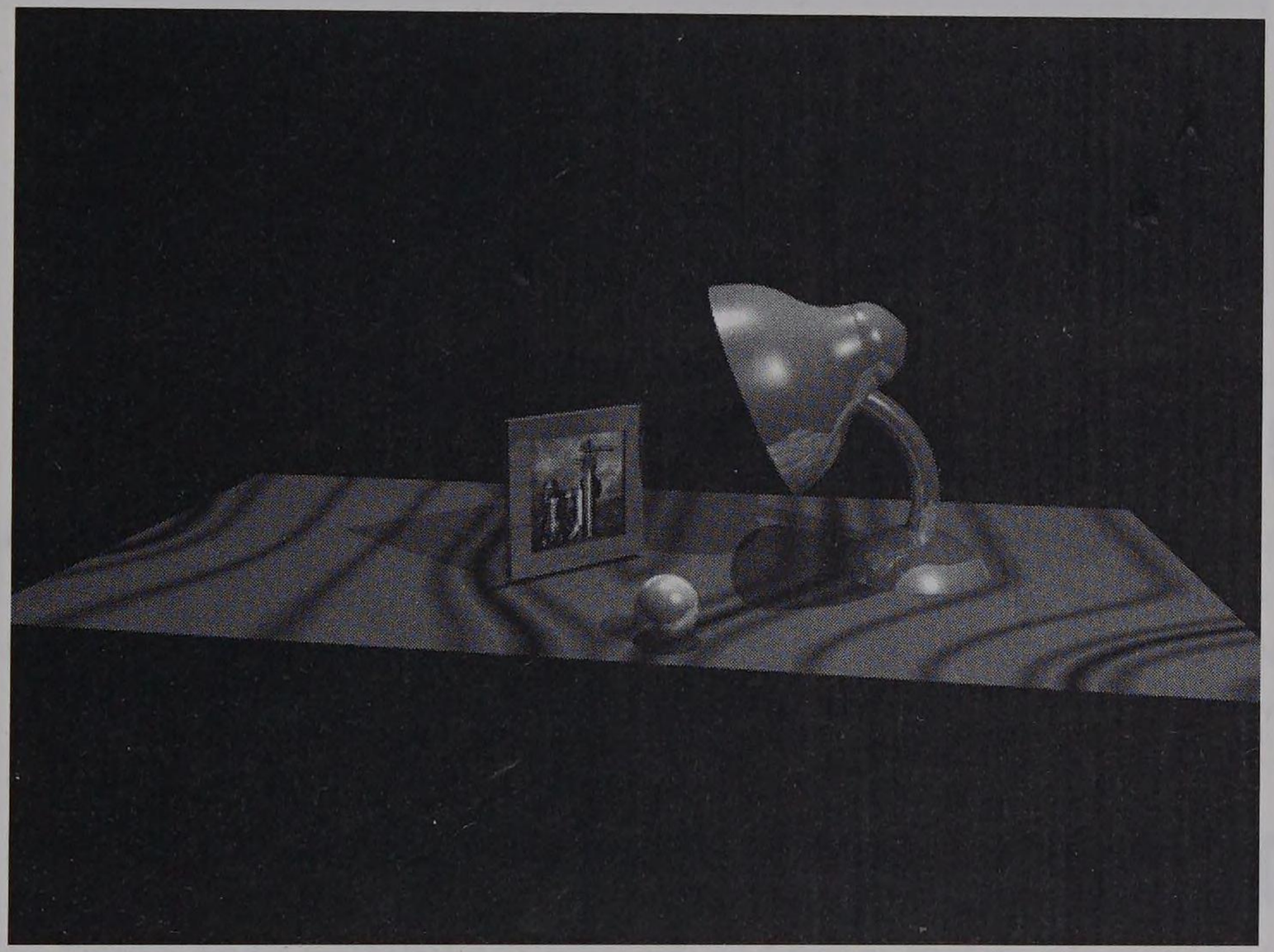


Figure 1.2
A ray-traced desk scene.

The second image (fig. 1.3) shows a box with a face. The surface finish is similar to that of the lamp in figure 1.2. The box is placed on top of a brown, flat plane. How many objects are there in this scene? You might be tempted to say three; however, the face alone consists of 807 triangles!

In the next section, you start with the basic concepts of ray tracing and work your way up.

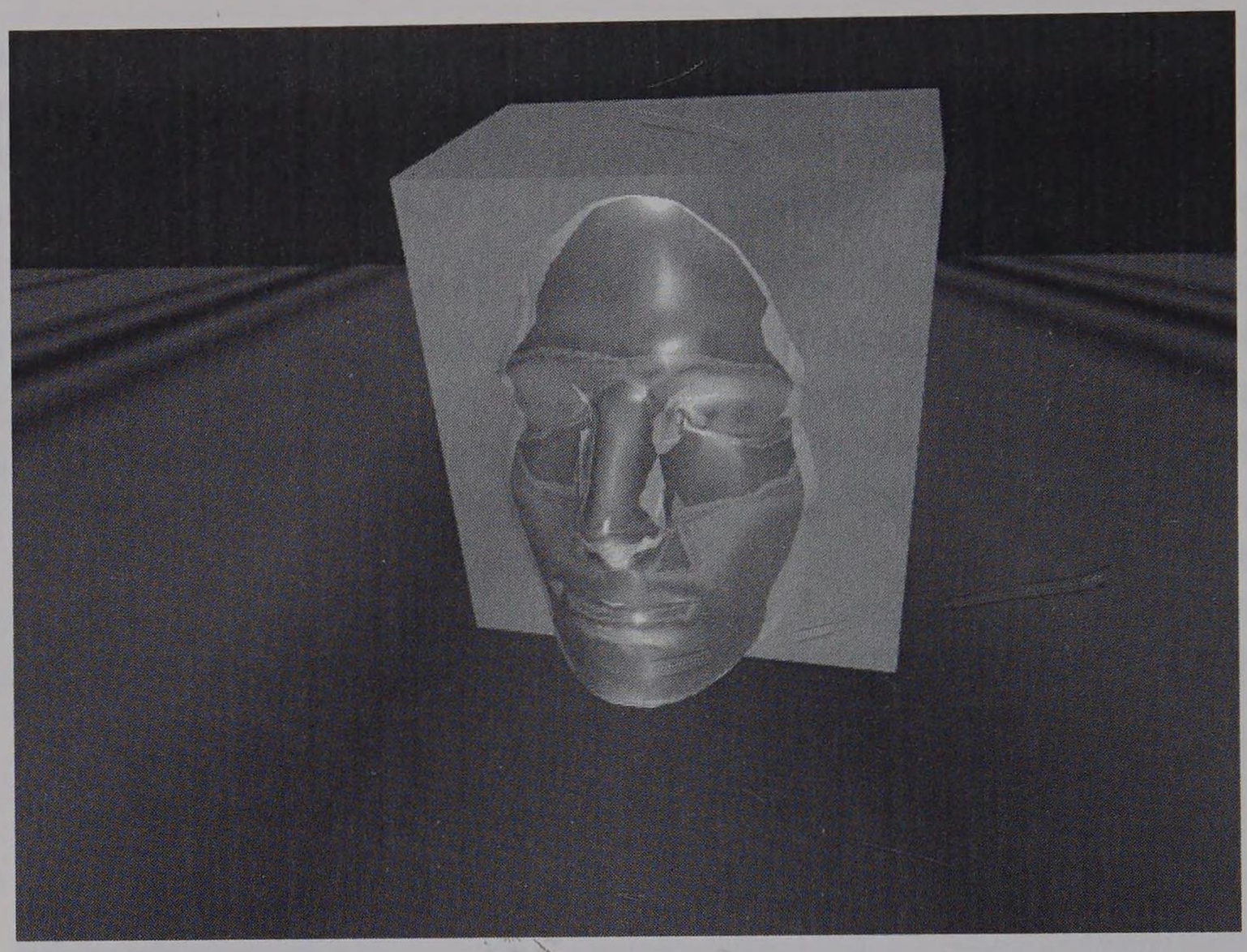


Figure 1.3

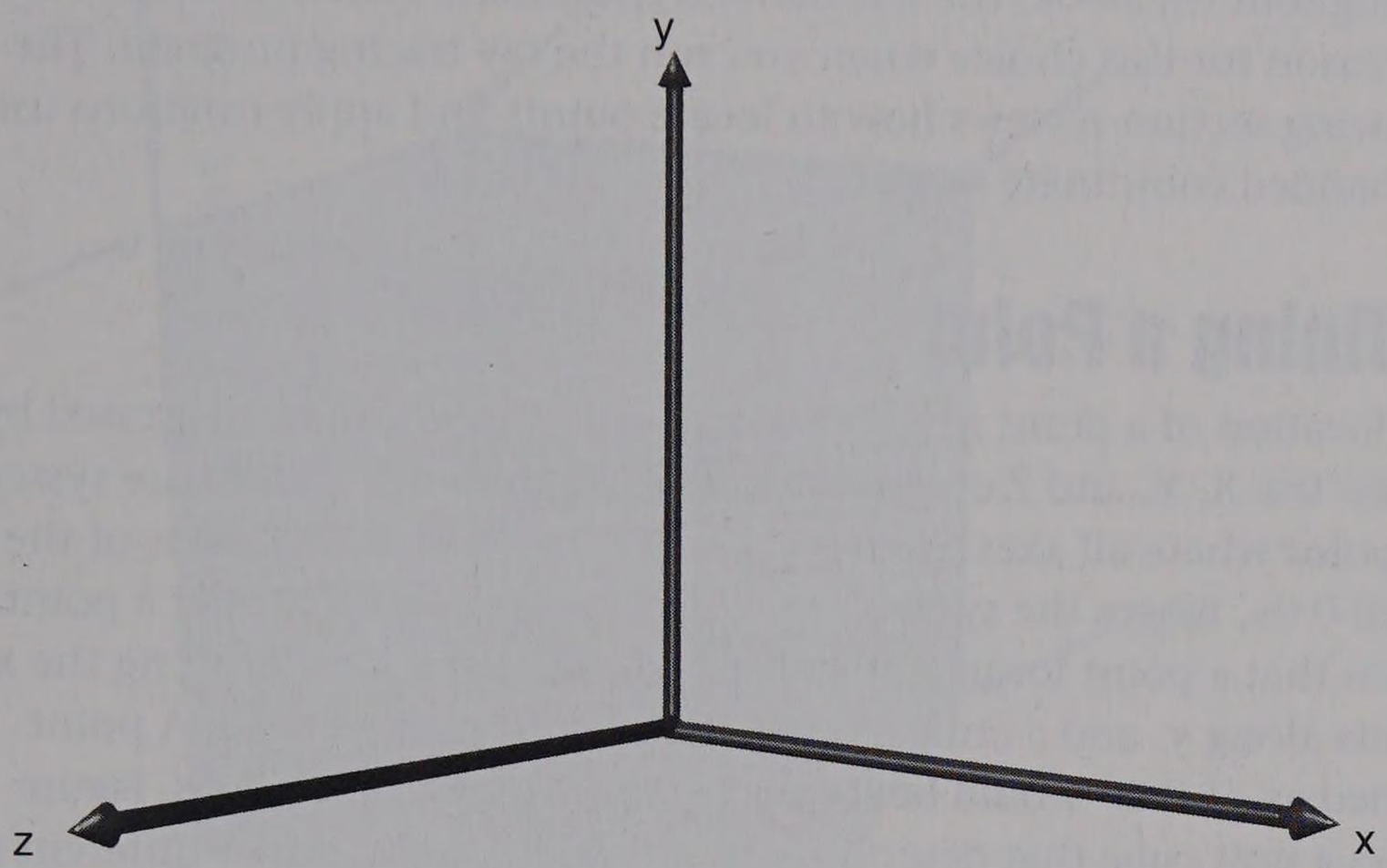
Ray-traced box with a face scene. (3-D model created with the HyperSpace 3-D digitizer from Mira Imaging, Inc.)

General Concepts

Before you create your first ray-traced image, you have to become familiar with some basic concepts. Later, this knowledge will help you control the creation of a photo-realistic scene. This section starts by describing how a 3-D world is defined mathematically and how to position objects in it.

3-D Coordinate Systems

A 3-D coordinate system is used to describe the location of an object in space. A 3-D coordinate system consists of three axes labeled *x*, *y*, and *z*, which are 90 degrees relative to one another. Two common orientation conventions used in coordinate systems are the *right-handed* and the *left-handed* conventions, as shown in figure 1.4. Note that the z axis is different in both coordinate systems. The direction of the rotations about a given axis also are different.



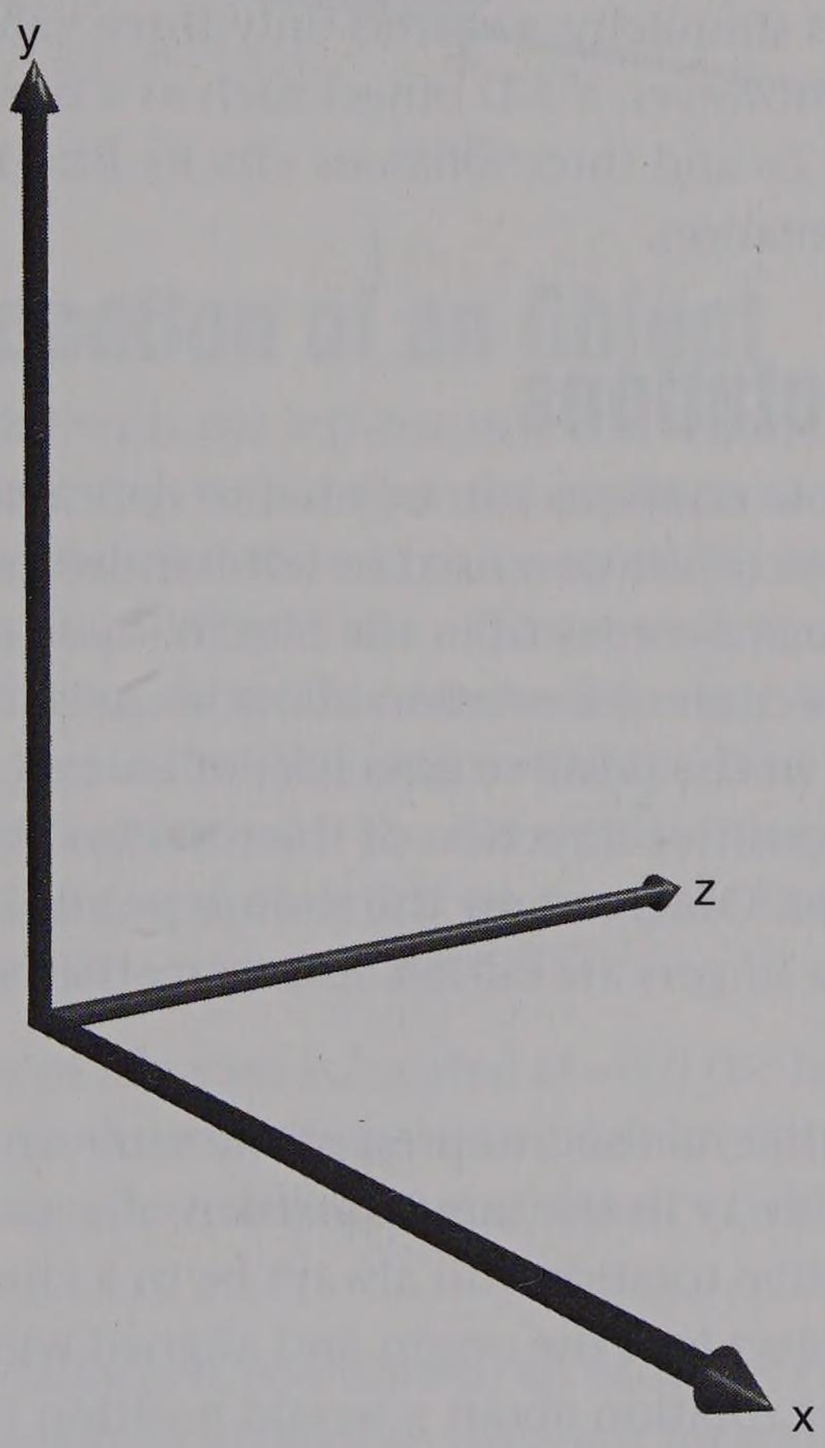


Figure 1.4

Coordinate system conventions.

Throughout the book, the left-handed coordinate system is used. You'll see the reason for this choice when you run the ray tracing program. The following section reviews how to locate points and apply rotations using the left-handed coordinate system.

Defining a Point

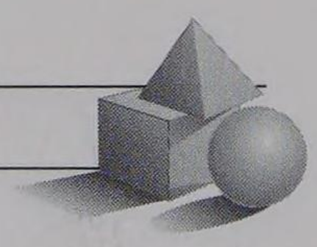
The location of a point in a 3-D coordinate system can be expressed by three values: the X, Y, and Z coordinates. The origin of the coordinate system is the point where all axes intersect one another. The coordinates of the origin are <0 0 0>, where the symbol <X Y Z> denotes coordinates of a point. This means that a point located at <a b c> has displaced *a* units along the x axis, *b* units along y, and *c* units along z, starting from the origin. A point defined as <1 0 0> would be located at the 1 unit in the x axis. Figure 1.5 shows a unit cube that describes the coordinate locations of different points.

A point, due to its simplicity, requires only three values or coordinates to be located in space; however, a 3-D object such as a box requires six: three coordinates <X Y Z> and three *rotations* <Rx Ry Rz>. First, you should know how to define a rotation.

Defining Rotations

To understand how rotations can be used to define the position of a 3-D object, look at the convention used in left-handed coordinate systems. The expression *left-handed* comes from the fact that you can use your left hand to express the direction of a rotation about an axis. By aligning the thumb of your left hand in the positive direction of an axis, the fingers of your left hand curl in the positive direction of the rotation about that axis. Figure 1.6 shows the concept. Observe how the thumb points in the positive direction of the X axis. The fingers are curled, following the positive sense of the rotation.

You can use another method to present the same concept. If you imagine standing very far away in the same direction of a given axis and looking at the origin, a positive rotation will always be in a clockwise direction. For example, if an object is at the origin and aligned with the positive y axis, a positive 90-degree rotation about x would position the object in the positive z axis. A negative rotation of 90 degrees from its new position would align it again with the y axis. Figure 1.7 shows this idea.



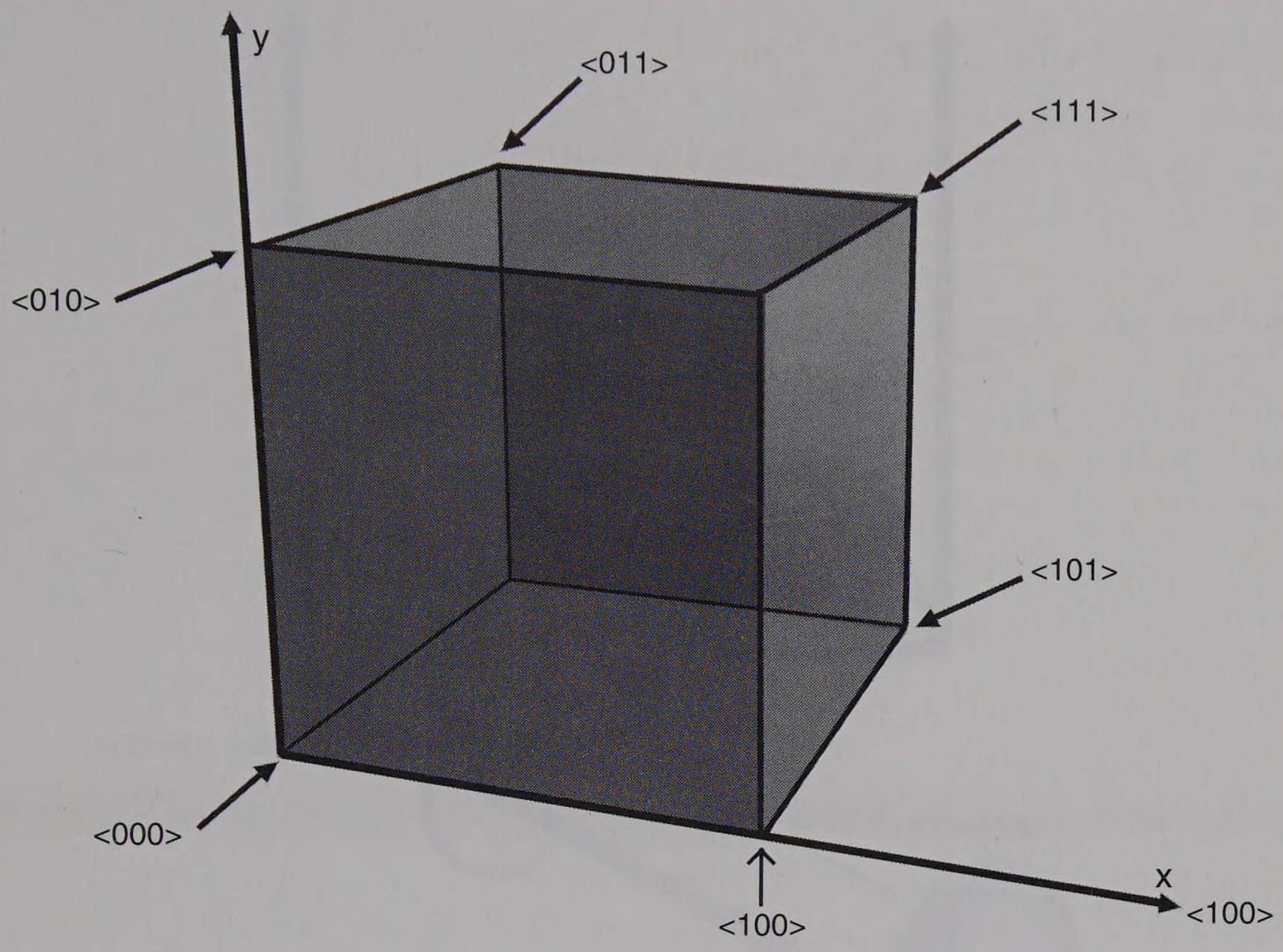


Figure 1.5
Point locations on a box.

Defining the Location of an Object

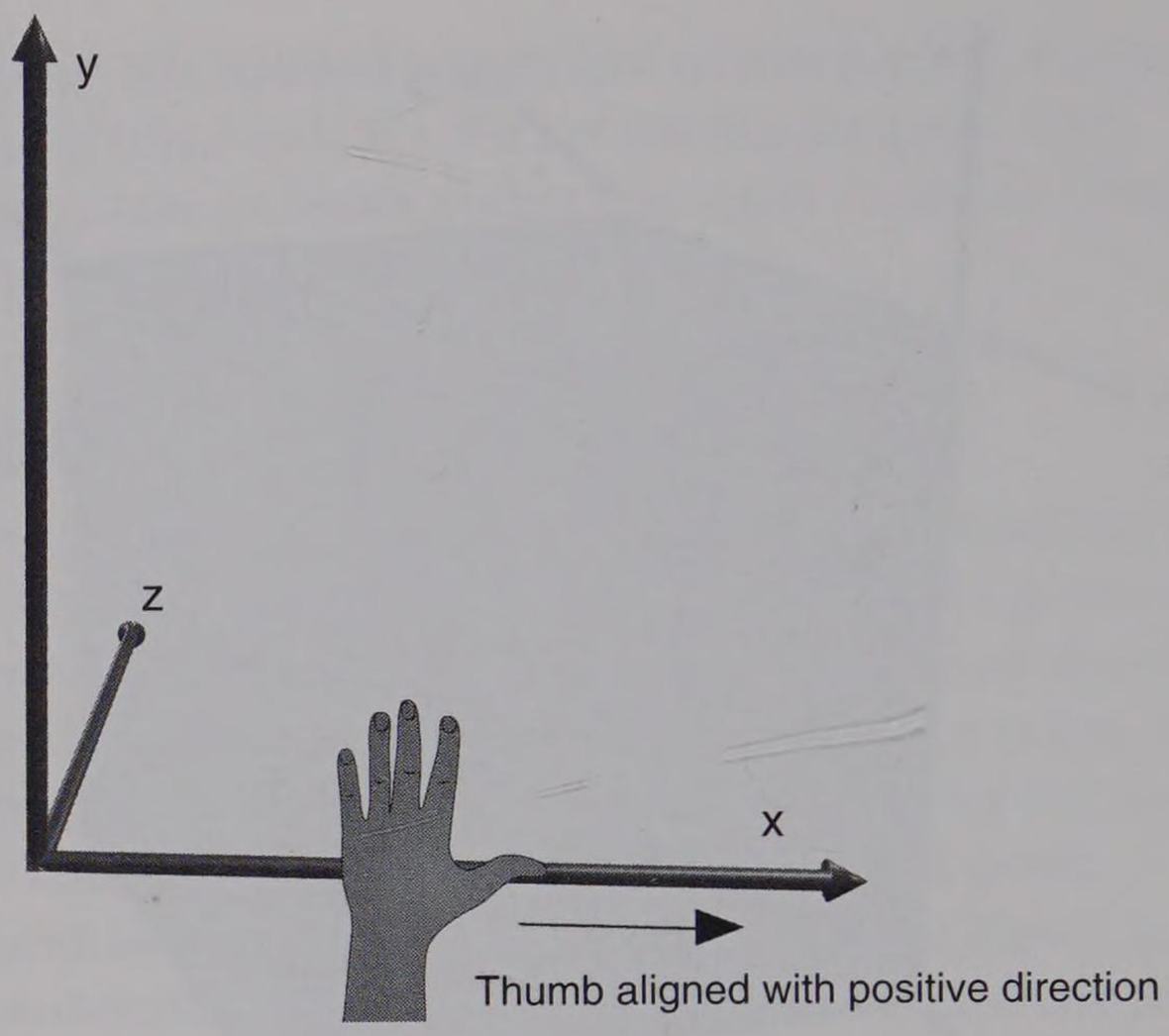
Now that you are familiar with the left-handed convention and how to define a rotation, the next step is to define the position of a 3-D object. As mentioned earlier, three coordinates and three rotations are required to describe the object's position. Why? Take a flower vase as an example. We want to locate the vase on top of a table where the origin of a coordinate system is. We decide to locate it at the origin <0 0 0>. Now a question arises: when we mentioned the position <0 0 0>, did we describe the orientation of the vase? Is it standing up? Lying down? If we compare both cases, the flower vase is rotated by 90 degrees (see fig. 1.8).

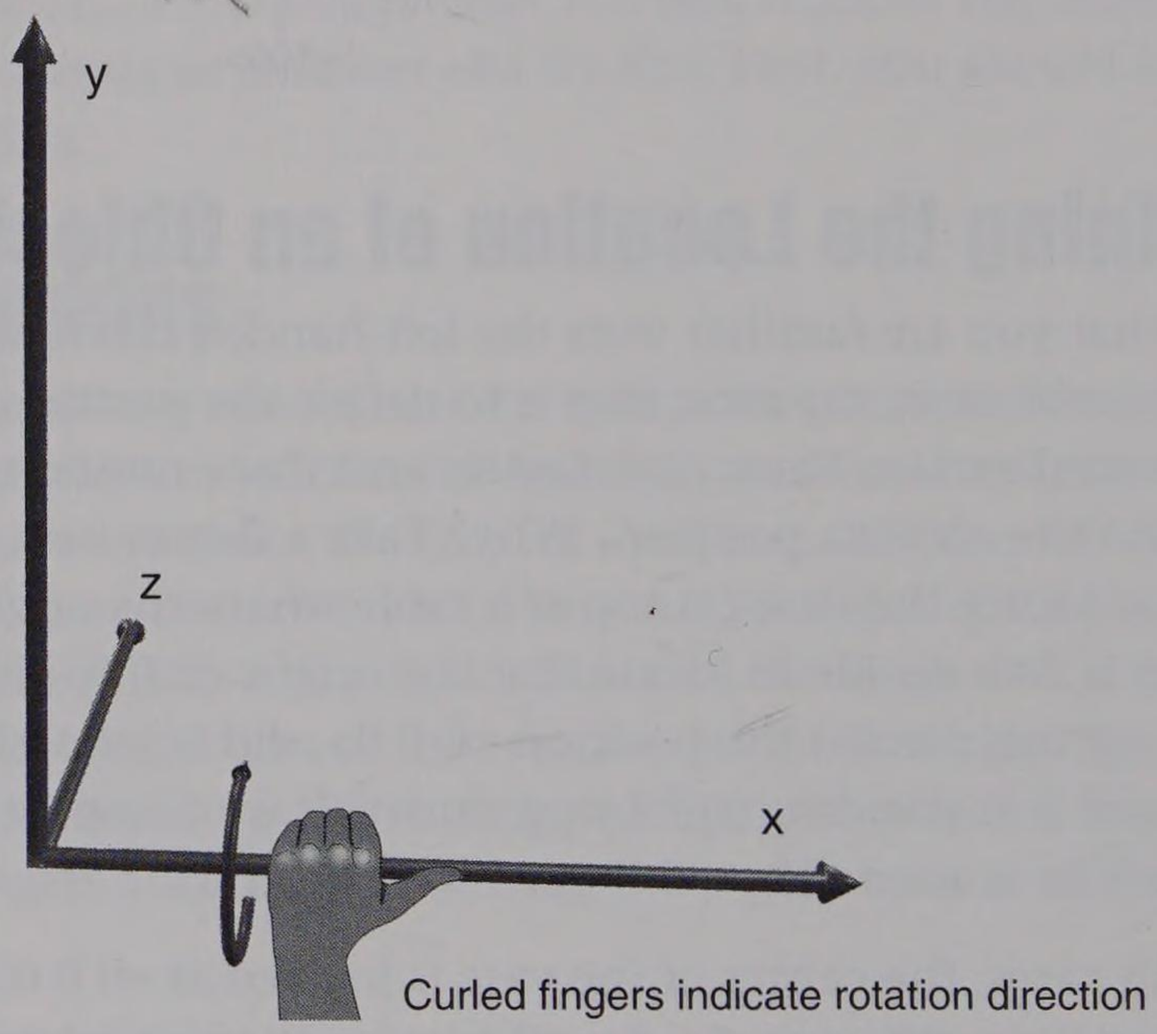
In both cases, the center of the vase is located at <0 0 0>; however, the orientation is different. To describe its position absolutely, you have to use coordinates and rotations rather than just coordinates (which is the method for describing points).

After an object or set of objects is positioned, an observer can "walk" through the scene and position himself in different locations to look at it. By moving around, different parts of the scene are exposed and what appeared to be hidden is now visible. This brings us to the viewpoint concept.

Figure 1.6

Using the left-hand rule.





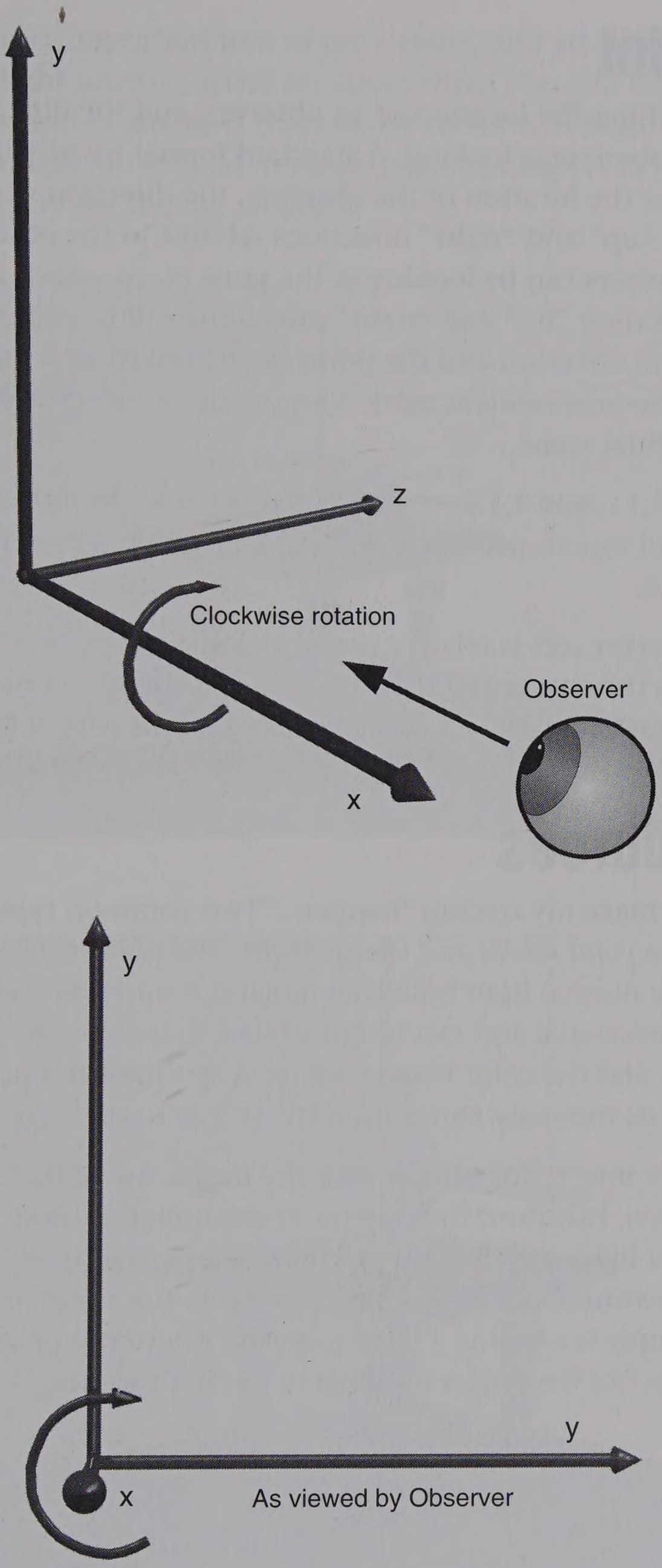


Figure 1.7

Positive rotation as viewed by an observer.

Viewpoint

A *viewpoint* defines the location of an observer, and the direction and point at which the observer is looking. A standard format usually includes information such as the location of the observer, the direction of the view, and where are the "up" and "right" directions relative to the observer (see fig. 1.9). Two observers can be looking at the same place—each in a different position—and their "up" and "right" directions will be different. The combination of direction and the point being looked at act as a field of view in the same way real cameras work. Varying these two creates a zooming effect on the final scene.

Figures 1.10, 1.11, and 1.12 present a scene viewed from three viewpoints: front, side, and top, respectively. The red and yellow spheres are used as an orientation aid.

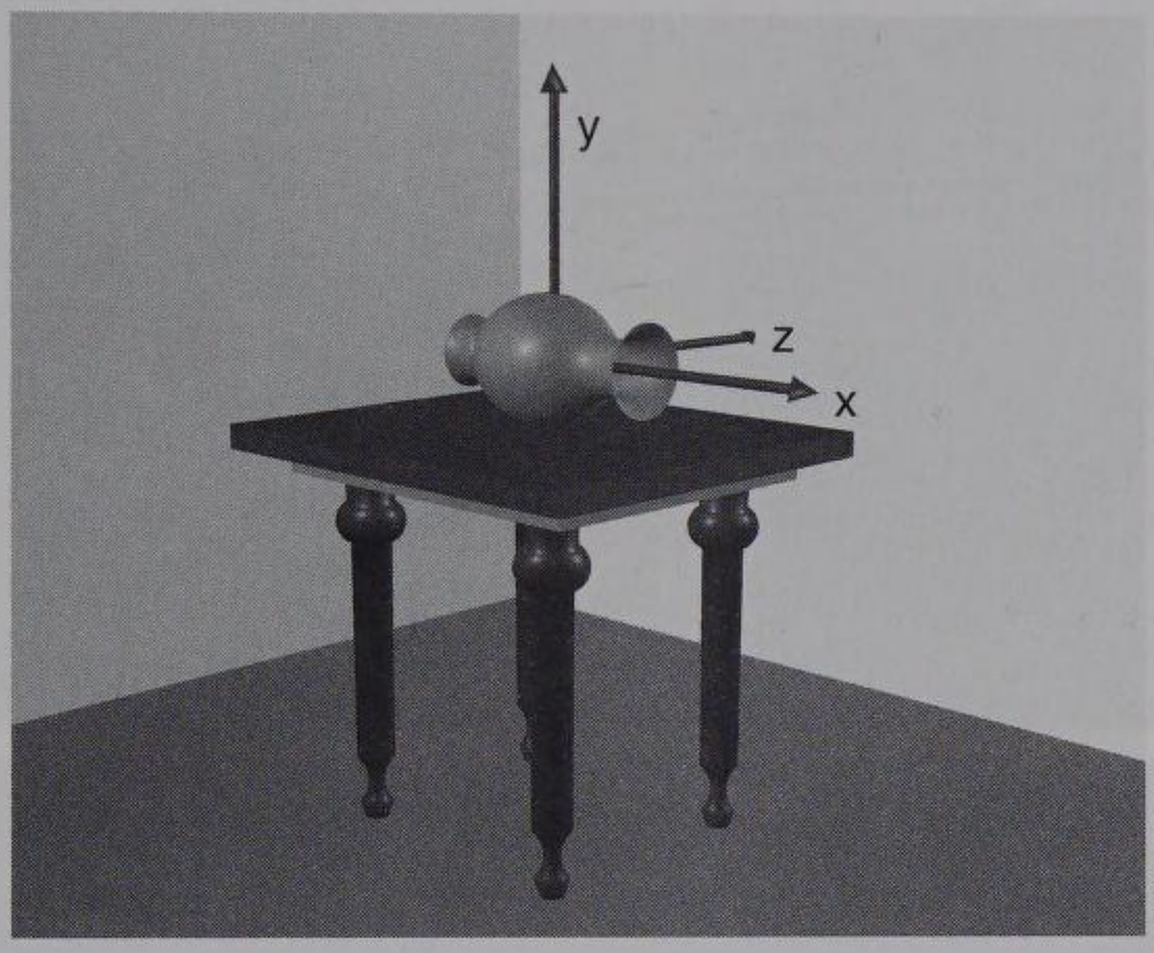
What the observer sees is relative to the location of the "eye" and also dependent on the amount of light that reaches the object or objects being looked at. Therefore, light sources are an important part of ray tracing.

Light Sources

Light sources make ray tracing "happen." Two common types of light sources are the *point source* and the *spotlight*. The point source can be compared to a normal light bulb that radiates light in all directions. The spotlight is directional and can be controlled. Light sources are described by their location and the color they produce. A spotlight has parameters that describe how its intensity varies from the center to the edge of the cone.

You can create interesting effects with the liberal use of light sources. (It is fair to warn you, however, that ray tracer computation time increases with the number of light sources.) An example of an interesting effect is the use of arrays of low-intensity light sources to create the effect of soft shadows. Another example is locating a light source at the center of an object to create the effect of the object itself being the light source.

Shadows are a very important part of ray tracing, and are usually a planned result of using light sources. There are times when you will be more interested in the creation of shadows than in the creation of highlights on an object. Light sources are studied in more detail in Chapter 2, in which we explore the Polyray ray tracing program.



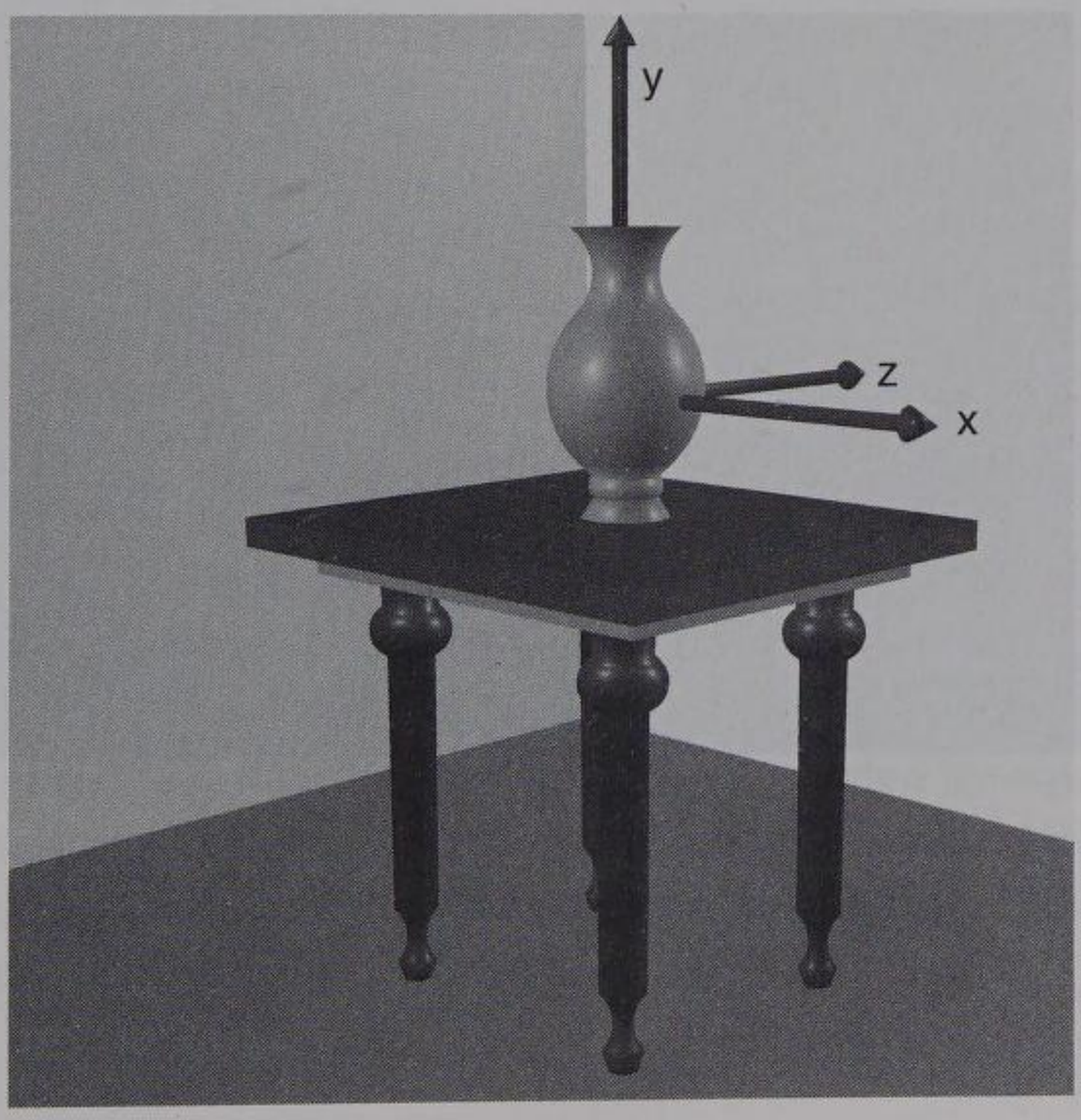


Figure 1.8
Objects located at the origin.

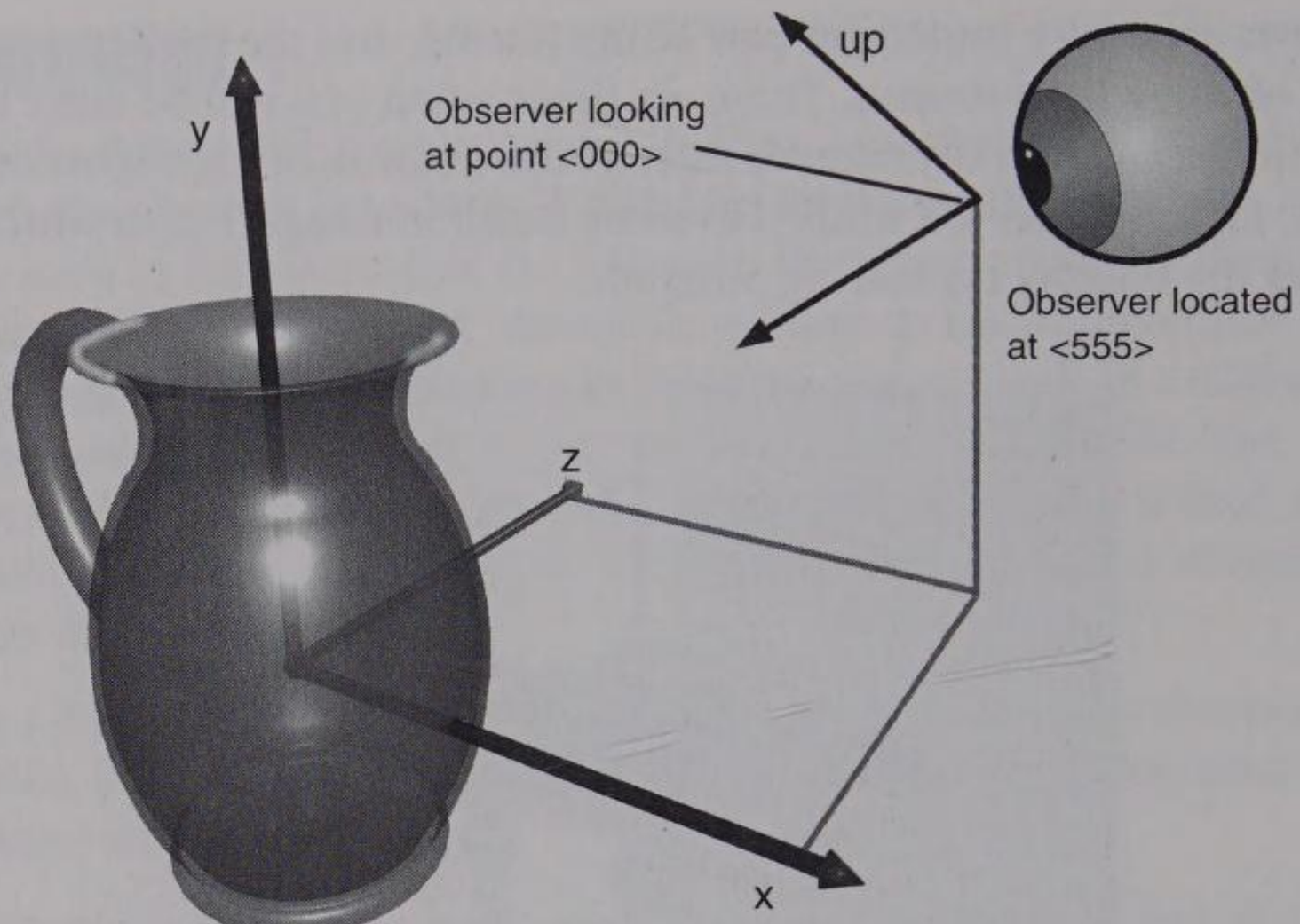


Figure 1.9

Defining an observer.

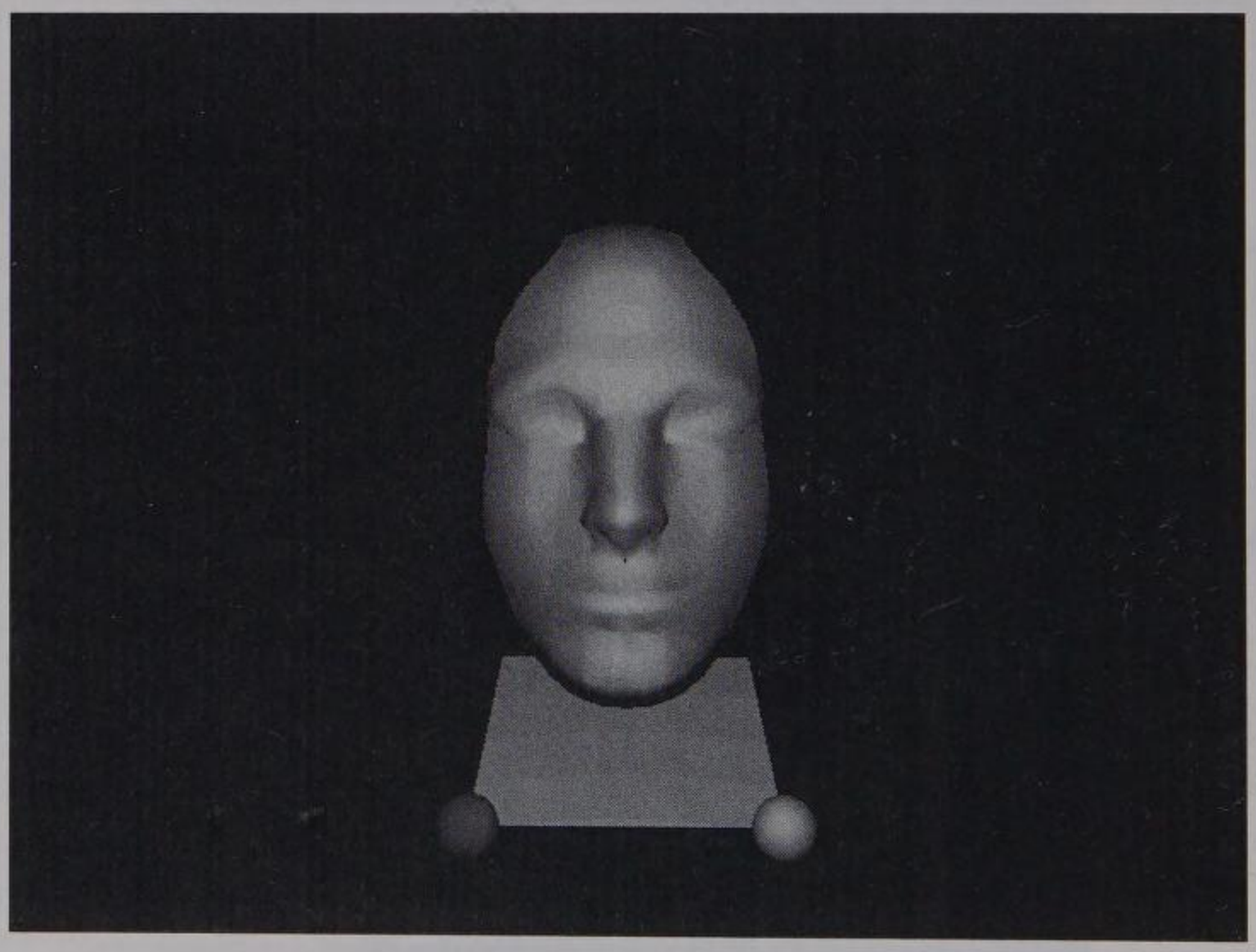


Figure 1.10

Front view of a face scene. (3-D model created with the HyperSpace 3-D digitizer from Mira Imaging, Inc.)

Figure 1.11

Side view of a face scene. (3-D model created with the HyperSpace 3-D digitizer from Mira Imaging, Inc.)

3-D Geometry

As mentioned earlier, ray tracing relies heavily on mathematically defined models. The objects a ray tracer uses are no exception. If you look around you, you can see that everyday objects can be modeled as a combination of simpler objects. For example, a table can be modeled as a long, thin, rectangular box with four cylinders as the legs.

Ray tracers usually have a standard set of 3-D objects defined and optimized for speed (see fig. 1.13). They also have other objects that have been defined in equation form. These equations have parameters that can be modified to generate different results. By varying the coefficients, a ray tracer can generate ellipsoids, hyperboloids, elliptic cones, and other objects. These objects tend to require longer computation time than other predefined objects such as the sphere.

There are other kinds of objects that cannot be generated by equations, and they require special techniques. Examples of such objects are lathe surfaces, extrusions, and rare undulating surfaces such as those used to create stylish and aerodynamic features in expensive sports cars.

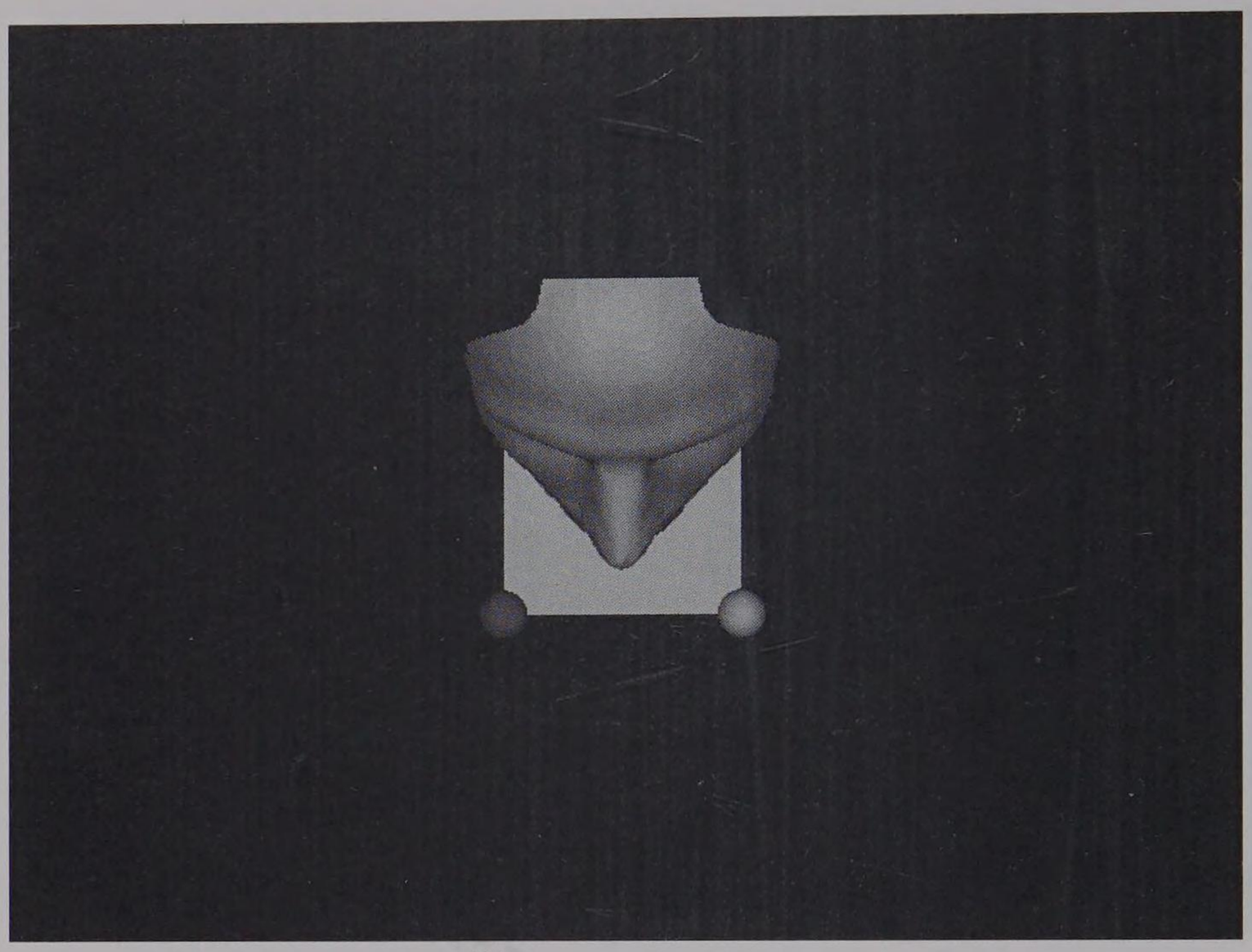


Figure 1.12
Front view of a face scene. (3-D model created with the HyperSpace 3-D digitizer from Mira Imaging, Inc.)

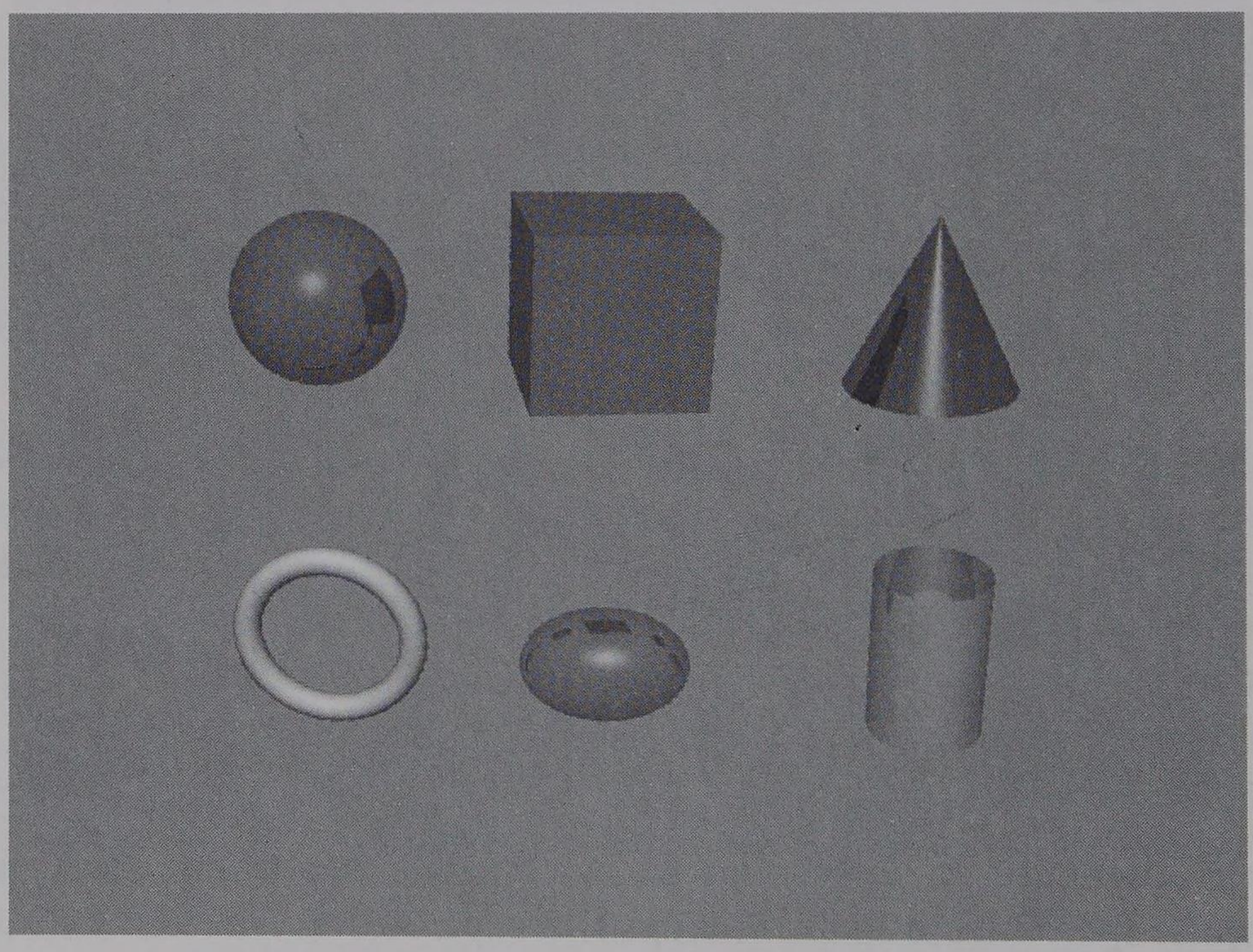


Figure 1.13
Examples of standard ray tracer objects.

Lathe Surfaces

Lathe surfaces are constructed by rotating a contour curve about an axis. These objects are also called *solids of revolution*. The surface is approximated by a collection of *slabs* or flat patches. The object appears as a faceted solid. The surface can be smoothed out by increasing the number of patches; however, this has the negative effect of increasing the time it takes to generate the image. Figure 1.14 shows the construction of a contour line and the final object.

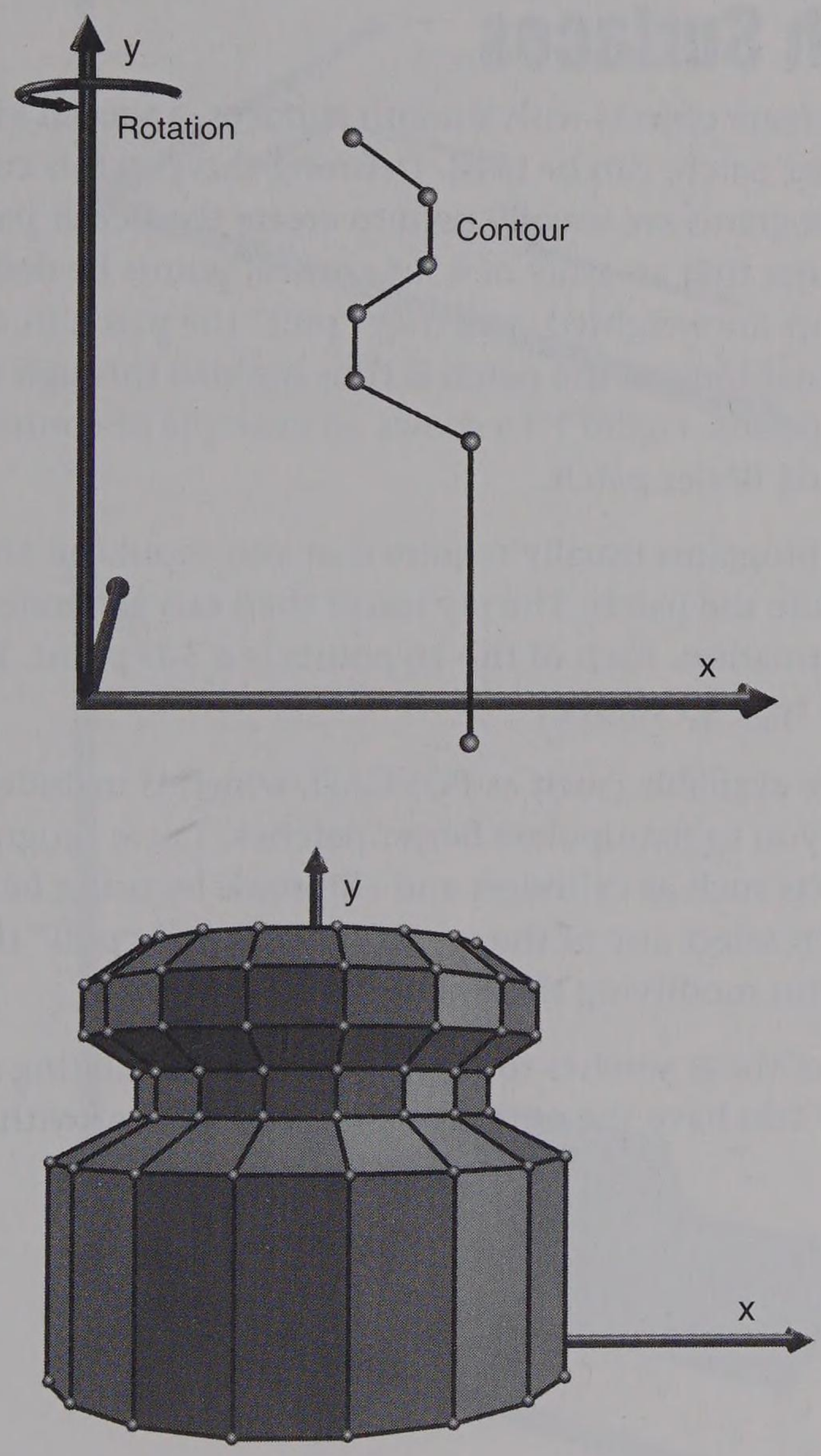


Figure 1.14
Constructing a lathe surface.

Sweep/Extruded Surfaces

Whereas a lathe surface is constructed by rotating a contour line about a given axis, an extruded surface, also known as a sweep surface, is created by translating the given contour curve along an axis. The new copied contour is then connected to the original one and patches are created. Figure 1.15 shows the creation of an extruded surface. Again, by increasing the amount of patches on the surface, you increase the smoothness.

Smooth Surfaces

In order to create objects with smooth surfaces, a special kind of patch, called a Bézier patch, can be used. Defining this patch is cumbersome; computer programs are usually used to create the Bézier patch. Bézier patches require that an array of 4 X 4 control points be defined. These control points are weighted, and they "pull" the patch in different directions. The final form of the patch is thus molded through the position of the control points. Figure 1.16 shows an example of control points and their corresponding Bézier patch.

Ray tracing programs usually require that you input the 16 control points in order to define the patch. The ray tracer then can generate the surface based on this information. Each of the 16 points is a 3-D point. In other words, you have to type 48 values.

Programs are available (such as POVCAD, which is included with this book) that enable you to manipulate Bézier patches. These programs usually create simple objects such as cylinders and ellipsoids by using Bézier patches. The user can then select any of the control points and "pull" the point in any direction, thus modifying the final form of the patch.

The ability of these patches to create complex undulating surfaces cannot be overlooked. You have the opportunity to experiment with Bézier patches in Chapter 3.

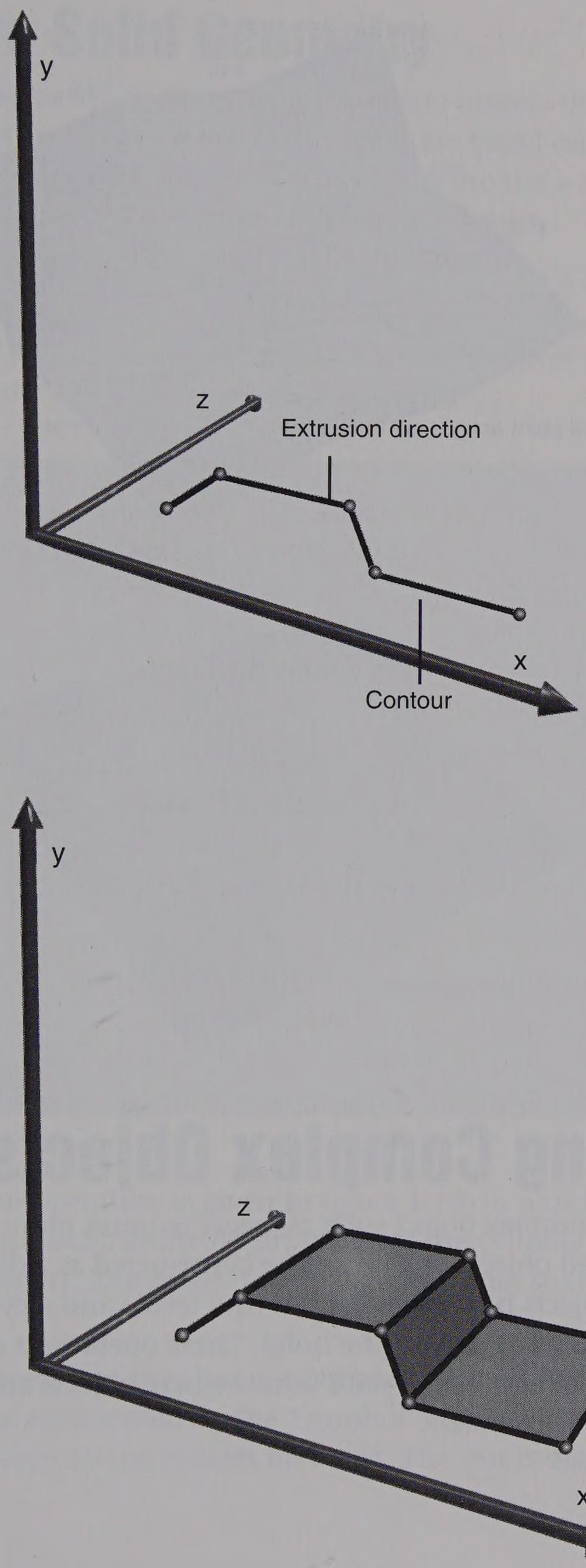
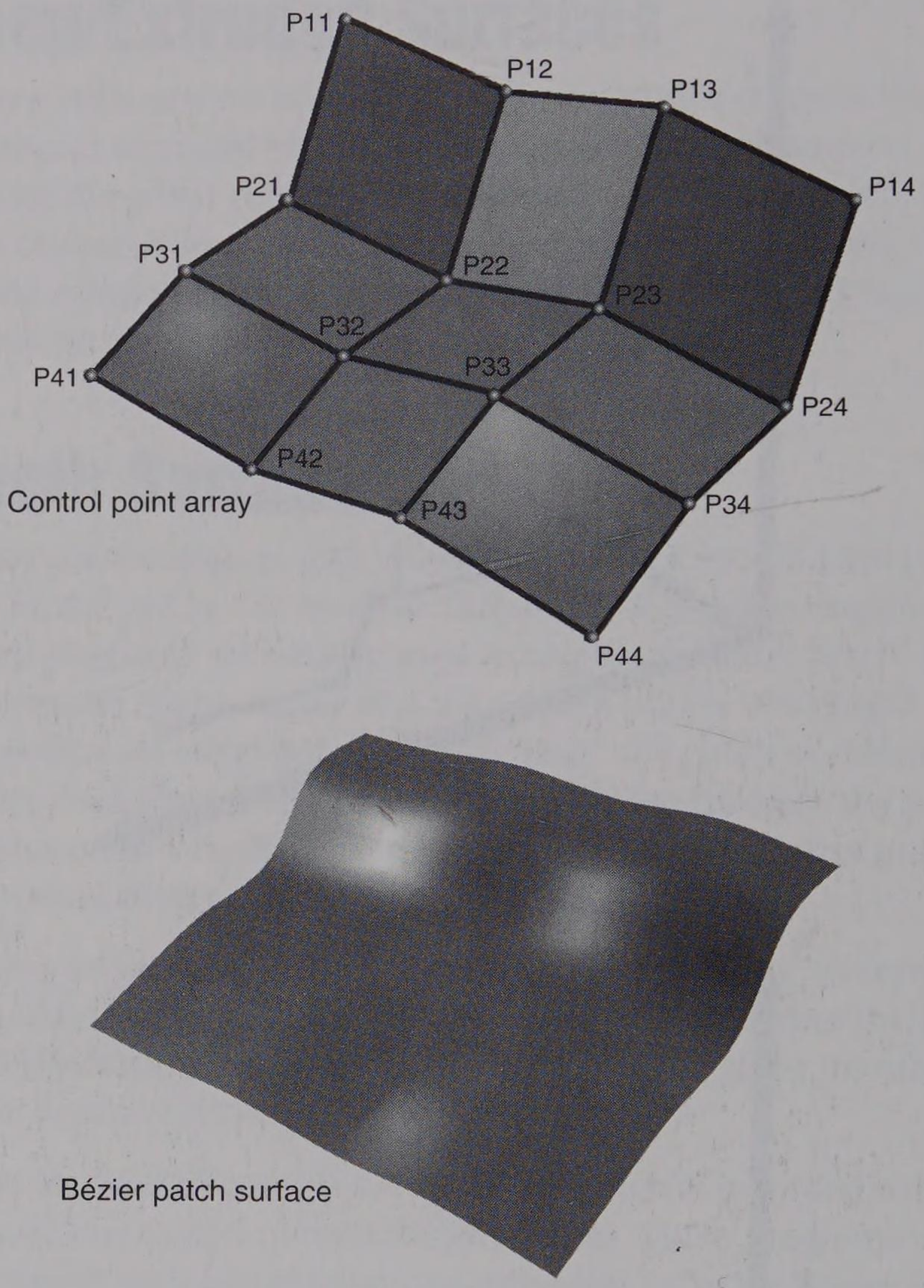


Figure 1.15
Constructing an extruded surface.

Figure 1.16

Example of a control point array and the final Bézier patch.



Building Complex Objects

Modeling a complex object such as a gear requires more than just using simply defined objects. A gear can be constructed as a disc from which prism-like objects have been cut out (the teeth) and a cylinder has been subtracted from the center (the hole). These operations of subtracting objects from other objects—and others where objects are added—are called *constructive solid geometry*.

Constructive Solid Geometry

Constructive solid geometry, or CSG, is used to create objects that are too complicated to be defined mathematically but that are based on adding or subtracting combinations of simpler objects. You can use the + and – signs to represent addition and subtraction operations, respectively.

Figure 1.17 shows a subtraction operation between a box and a cylinder. Observe that the operation is (box – cylinder) and *not* (cylinder – box). The result is a box with a hole in it. The subtraction operation gives a different result depending on the order of the objects.

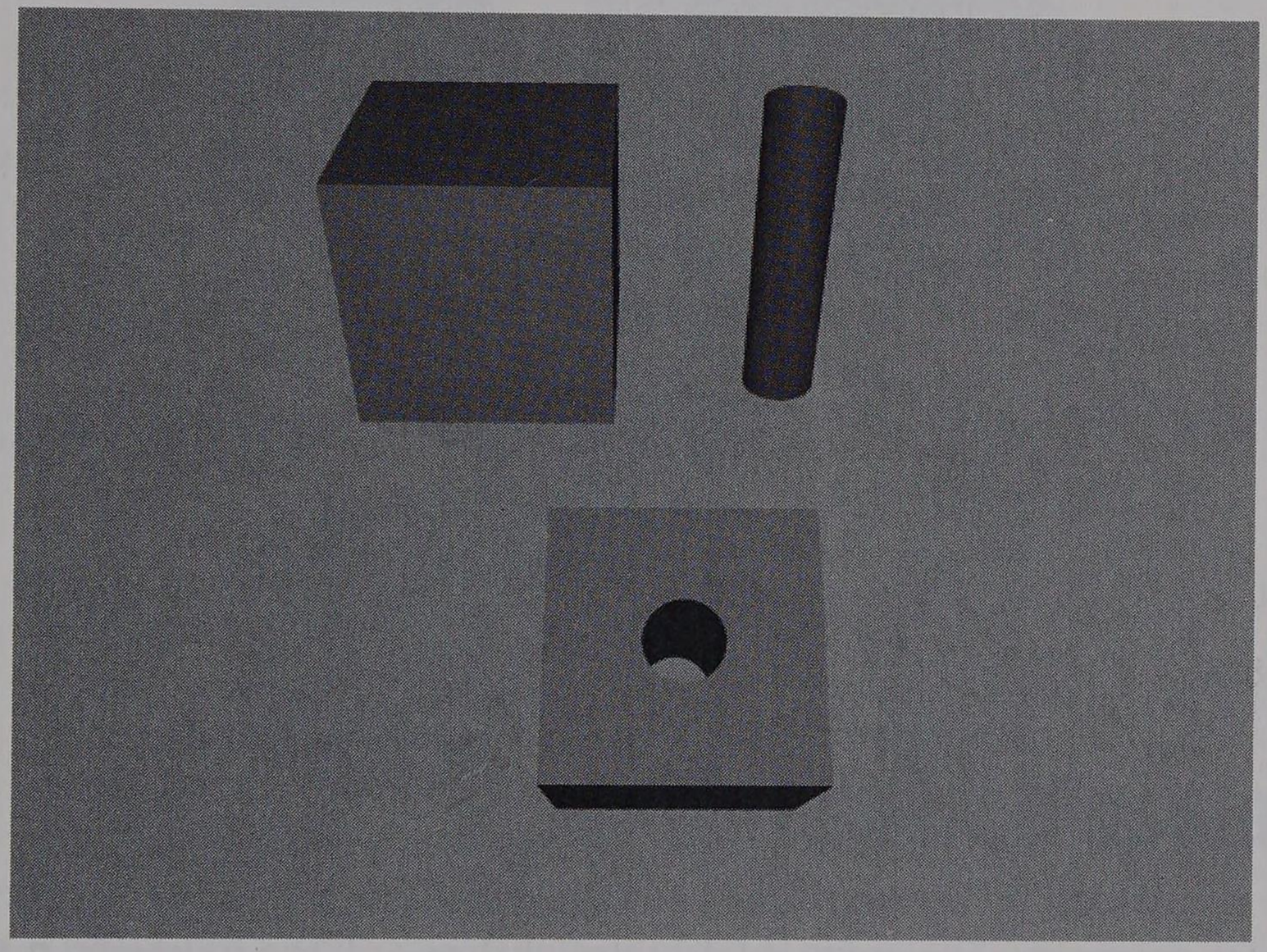


Figure 1.17

A difference operation—the cylinder is subtracted from the box.

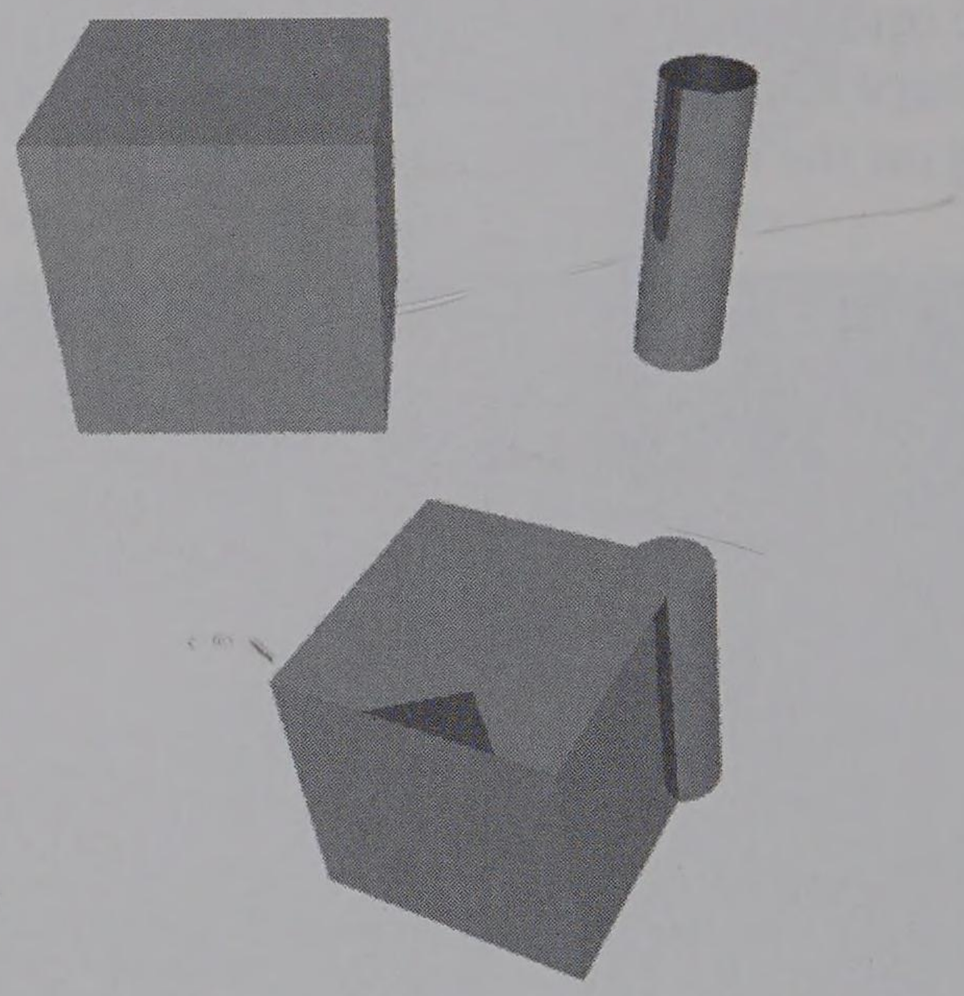
An example of a union operation is given in figure 1.18, in which another box is used. The box is placed at the origin, whereas the cylinder is placed at one of the box's corners. The result is a box with a corner that forms a cylinder.

There is yet another operation that has not been mentioned. It is called *intersection*, and can be represented by the * symbol. This operation keeps what is common between all the objects involved. The rest is discarded.

We have been using the symbols +, –, and * to describe these operations. These operations are called *Boolean operations*. Although some programs use the mentioned convention, others simply use explicit keywords to describe them: `union` (+), `difference` (-), and `intersection` (*). The operations are the same. Figure 1.19 illustrates each operation.

Figure 1.18

An example of a union operation



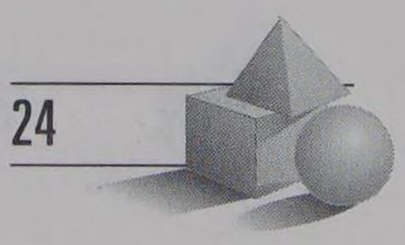
Boolean Operations

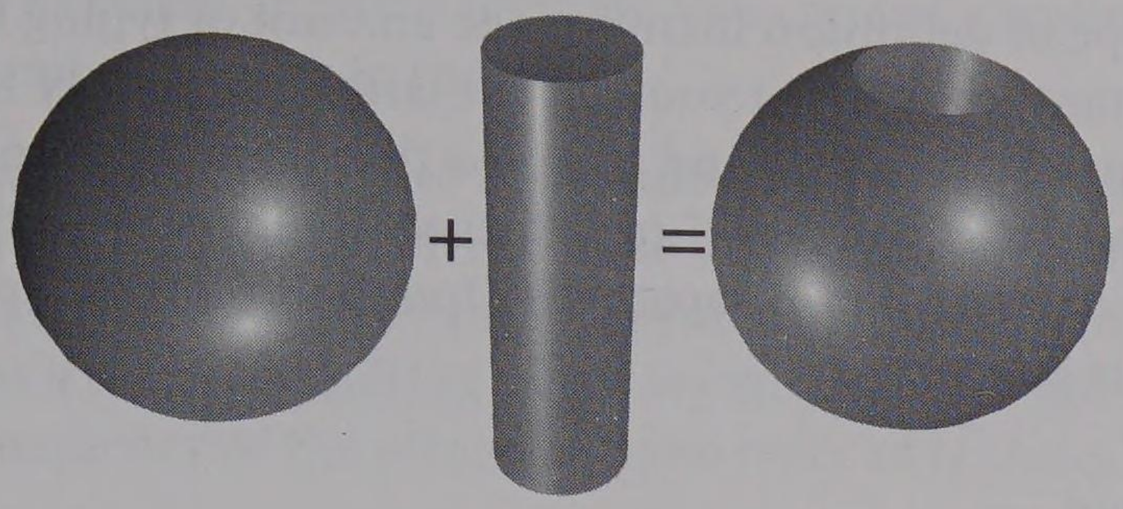
Some ray tracers use keywords for Boolean operations (CSG) to construct complex geometry. The usage is fairly straightforward; the following is the format:

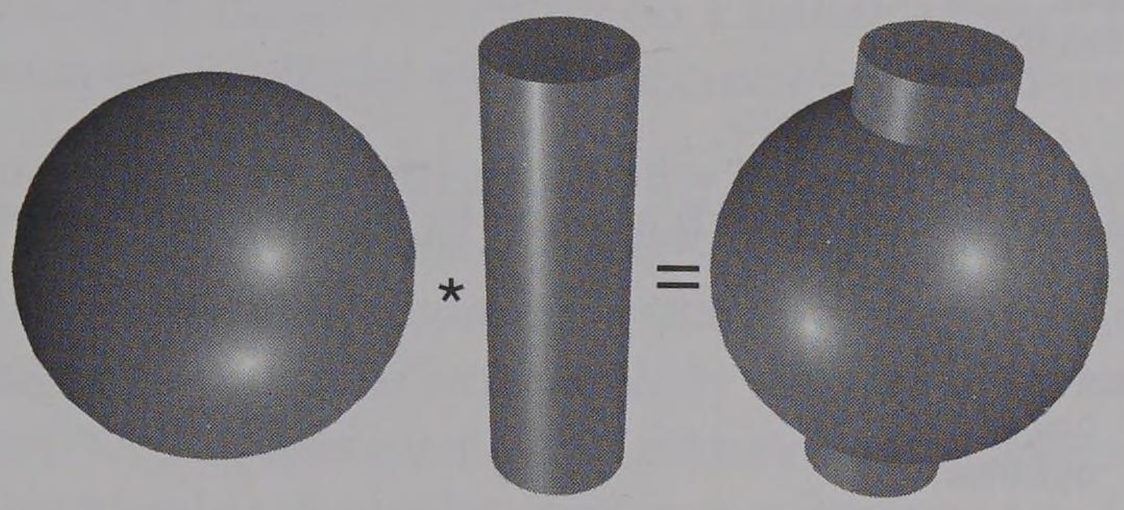
```
Boolean operation name { first object  second object .... nth object}
```

The following are explicit examples:

```
union { box cylinder  }
difference { box cylinder}
intersection { sphere cylinder }
```







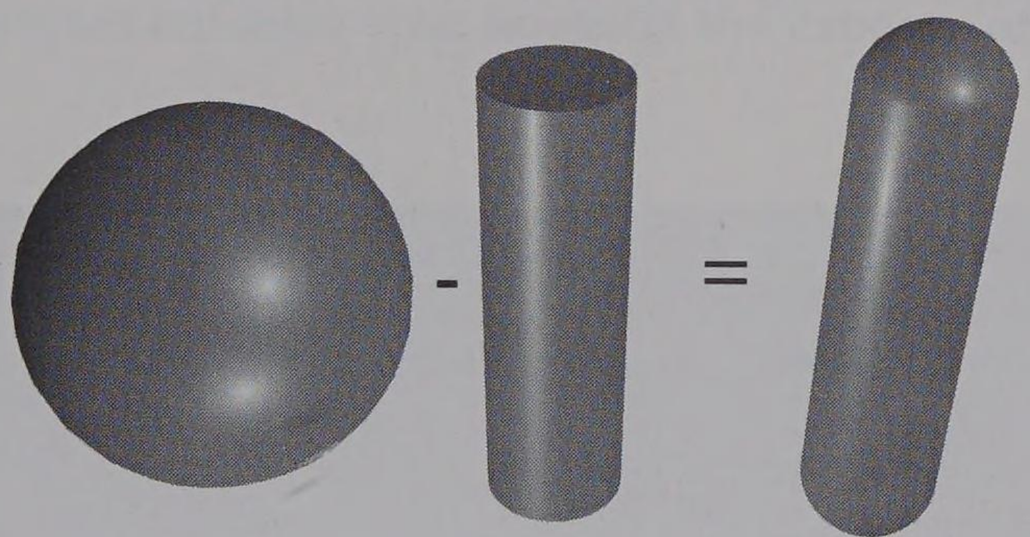


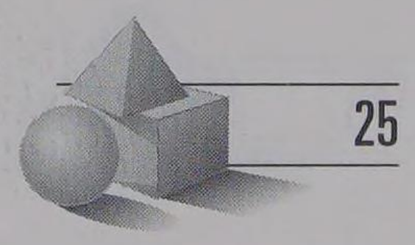
Figure 1.19
An example of an intersection operation.

Boolean (CSG) operations also can be nested to create more complex definitions, as in the following:

```
union {
              sphere
              difference {
                  box
                  cylinder
              }
}
```

In mathematical form, this operation looks like this:

sphere + (box – cylinder)



Because this type of definition increases the amount of typing required, another important feature that you will use later is the ability to attach names to groups of operations and then use that name to simplify the creation process. This "attaching" of names to groups of objects and operations is similar to some macro capabilities found in most PC applications. The following is an example:

```
union {
      ellipsoid
      difference {
            box
            cylinder
      }
}
```

To simplify this operation, attach a name to the difference operation, as follows:

```
MyObject = difference {
            box
            cylinder
          }
```

Now that a name has been given, rewrite the example:

```
union {
     ellipsoid
     MyObject
}
```

This expression is identical to the one given previously. Throughout the book, we use the symbols +, –, and * rather than the keywords to describe Boolean operations.

Adding Photo-Realism

There are various mechanisms by which you can increase the realism of a ray-traced scene. The color of a surface can be modified by varying the red, green, and blue contents. The surface can be made transparent or opaque. You can give a surface mirrorlike qualities by modifying the reflection and roughness values, or you can make it metallic by changing the brilliance value.

Most ray tracers include different types of texture models, such as wood, agate, marble, and checkered. You can use these as-is, or modify them to create the effect that you are looking for. Their effects can be increased or

decreased. For example, in wood textures, you can use a scaling factor to change the number of rings in the object.

Perturbation techniques are used to add randomness to textures and are used to create such great effects as waves, ripples, or bumps on surfaces. The surface is not really rippled; however, the intensity of the light at each point is calculated as if it were, resulting in a wavy surface. You can easily modify the size and frequency of the effects of these types of textures.

Mapping textures are used to "paste" an image on a surface. For example, you might want to map an image of a clown on a sphere, or map the image of your favorite soda on a cylinder when you create a kitchen scene.

Another technique to improve image quality is called *antialiasing*. Antialiasing is used to smooth out an image by reducing the "jagged" look (see fig. 1.20). If you have used computer drawing programs, you have been aware that straight lines do not look perfectly smooth; they have rough edges or "jaggies." Antialiasing is used to eliminate the jaggies. This is accomplished by filtering the image data as it is generated or by studying a given pixel or point on the screen in more detail (*over sampling*). Needless to say, these techniques enhance your image at the expense of computer time (see fig. 1.21).

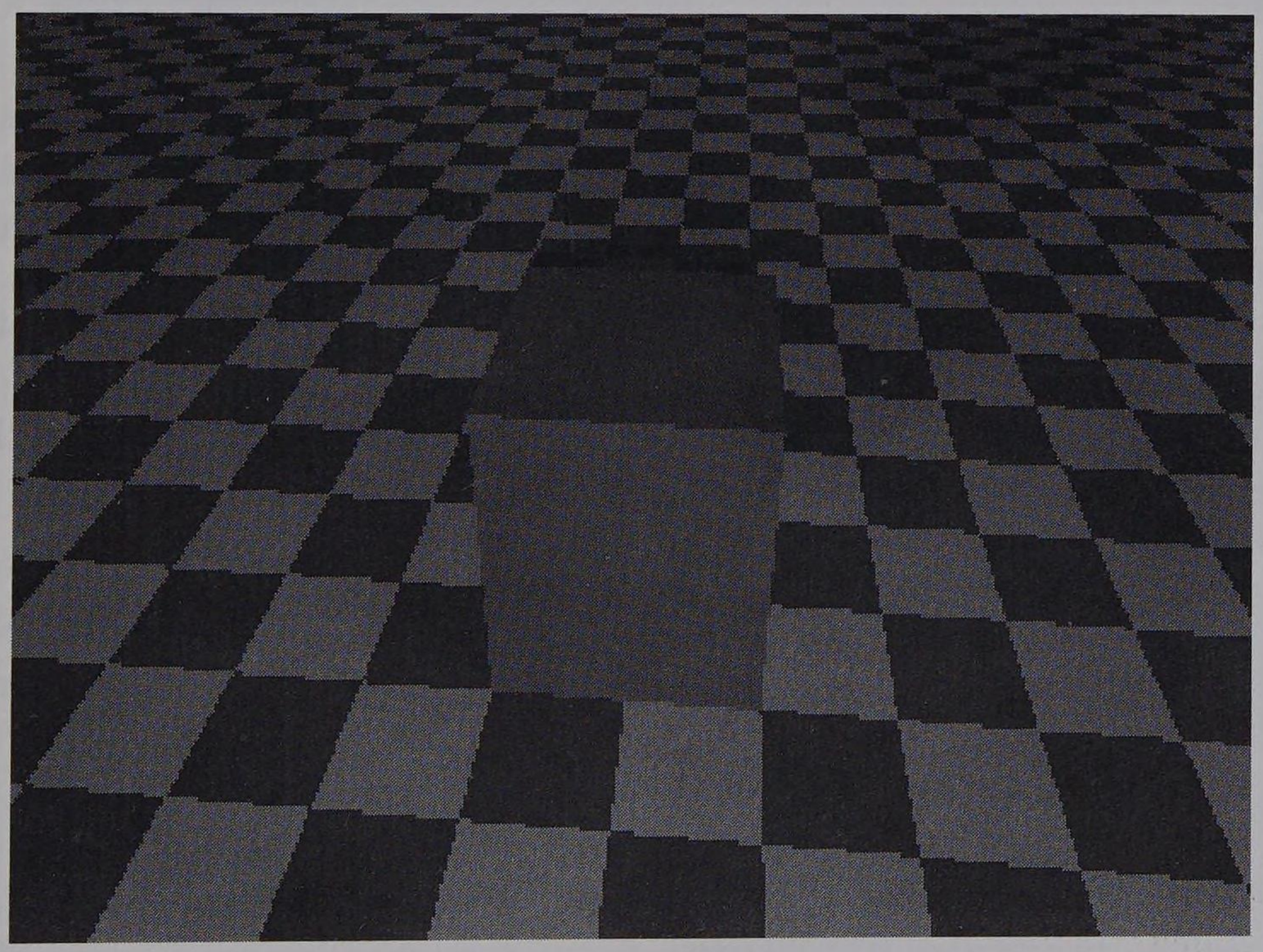


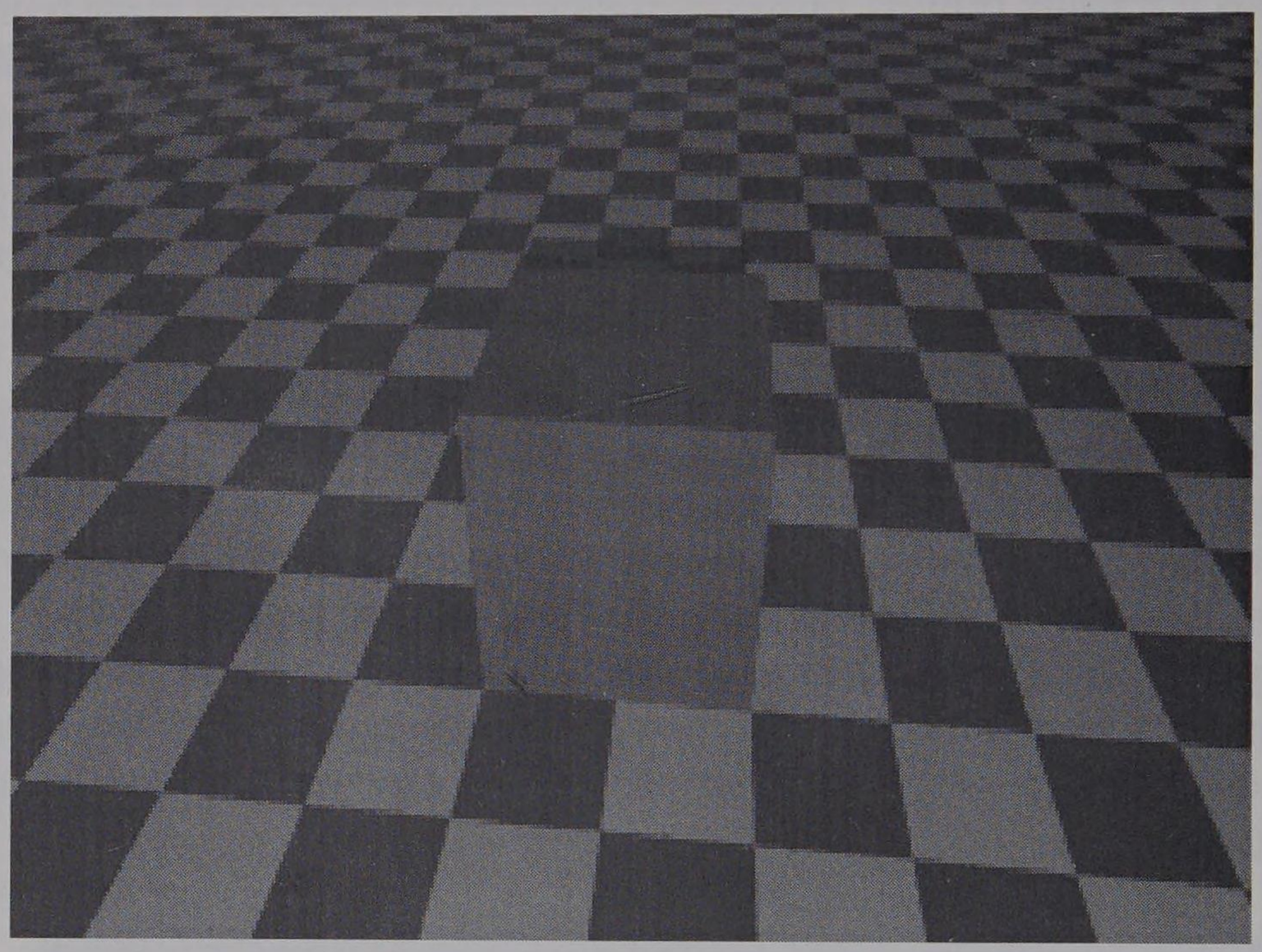
Figure 1.20

An image without antialiasing.

There are many other interesting features and more in the ray tracing program included with this book. In the next chapter, you get your first look at Polyray, an excellent ray tracer that will help you get started with photo-realistic image creation.

Figure 1.21

An image with antialiasing.



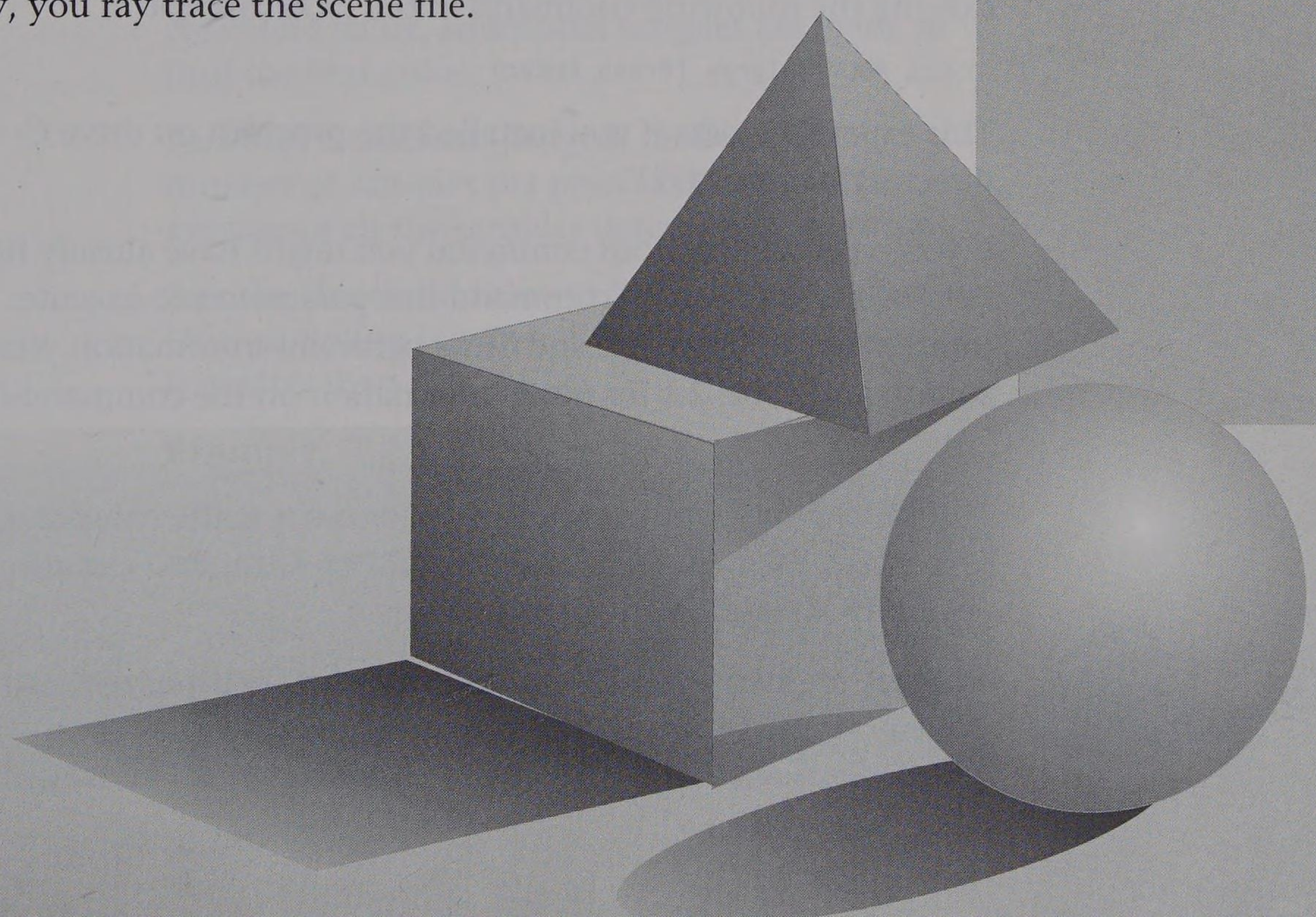
Summary

In this chapter, you learned what ray tracing is and the basic concepts necessary to develop a scene file. You studied what a left-handed coordinate system is and how to define points and 3-D objects in it. You became acquainted with different standard 3-D objects. You were presented with the concept behind Boolean operations or constructive solid geometry and its importance in building more complex objects.

The importance of texture used to create realism was described. Also described were the different effects that can be created, such as metallic and marble finishes, waves, and bumps. In the next chapter, you have the chance to develop your first ray-traced image.

Your First Ray-Traced Image

Chapter 1 introduced you to basic ray tracing concepts. This chapter gives you the opportunity to create your first scene file and ray trace it. You also are guided through the development of a simple scene file. After you create the file, you will "dissect" it and study its parts. Finally, you ray trace the scene file.



The Polyray Ray Tracer

Polyray is an excellent ray tracer that enables you to create image files by using its database of standard primitives such as the ones mentioned in Chapter 1 (spheres, cones, boxes, and so on). It also enables you to create your own objects by using mathematical functions—such as z = f(x,y). All of the features that were presented in the previous chapter and more are available. As you progress in the book, you will become familiar with Polyray's features.

Polyray was developed by Alexander Enzmann, using as a starting point the source code from another ray tracer called MTV, written by Mark VandeWettering. The MTV ray tracer is in the public domain. Although the code for MTV is freely available, Polyray is a shareware program.

Using Polyray

Let's start getting acquainted with Polyray by learning how to run the program from the DOS prompt and understanding which command-line parameters are available.

Running POLYRAY.EXE

To start Polyray, move into the directory containing Polyray, and simply execute the following command at the DOS prompt:

```
C:\ POLY>polyray [Press Enter]
```

This example works if you installed the program on drive C: under a directory named POLY.

If you typed the previous command you might have already figured out that Polyray requires a set of command-line parameters to execute. The parameters indicate which file to use and other pertinent information. Read the next section and Table 2.1 for more information on the command-line parameters

NOTE: Polyray requires an IBM PC or compatible computer with at least a 386, a VGA graphics card, and 2M of RAM. You can run it from DOS or a DOS window inside Windows.

There are two versions available for those with or without coprocessor. Refer to the Introduction for more details.

Command-Line Parameters

The following is a list of the command-line parameters Polyray accepts. It is not practical to type them every time you want to ray trace a scene file, so I suggest you create a batch file to execute Polyray. Normally, to run Polyray you should use the following format for the command:

```
polyray.exe filename.ext [rest of command-line parameters]
```

in which filename.ext is the name of the scene file. Table 2.1 describes in detail the rest of the command-line parameters. Don't worry about memorizing all the Polyray options; you can refer to this table when you need help.

Table 2.1. Command-line parameters.

Parameter	Action
Antialiasing Parameters	
-a	Performs a simple antialiasing method of averaging the neighboring pixel colors. Degradation of the image will occur due to the averaging process.
-A	Performs an adaptive antialiasing method. It is dependent on the threshold value (parameter -T) and the number of samples per pixel (parameter -S). This method samples a pixel and its neighbors. If the color difference is greater than the threshold value, additional samples are made to find the best color.
-J	Performs a random (jittered) method. The number of samples per pixel is constant. The average of all the samples determines the color.
-T n	Sets the threshold value to force over sampling. Over-sampling is active if the difference in colors is greater than n. **Example:** -T 0.1
-S n	Sets the number of samples per pixel (n) for the adaptive (parameter -A) method.

continues

Table 2.1. Continued.

Parameter	Action
File Output Operations	
-b n	Saves the image data after calculating n pixels. **Example:** -b 1280
-B	Saves the image data after calculating every line. Use this parameter if you are concerned with saving as much data as possible in case something goes wrong.
File Output Operations	
-o f	Specify the output file name f. If the parameter is not used, OUT.TGA is assumed. **Example:** -o table.tga
-p n	Set the number n of bits per pixel on the output image file. Acceptable values are 8, 16, 24, and 32. **Example:** -p 24
-u	Use uncompressed TARGA format on the output file.
Image Quality and Size	
- r m	Select method m to generate the image (the lower the number, the better the quality). **Value** **Method** 0 ray trace 1 scan convert (requires substantial amounts of memory) 2 wireframe (to screen only)
-d p	Dither objects with probability p (the number must be between 0 and 1). Use this parameter to reduce image generation time. Quality is also reduced.

Example: -d 0.5

`-D p` Dither p percent of rays used (the number must be between 0 and 1).

Use this parameter to reduce image generation time. Quality is also reduced.

`-P n` Use palette number n.

Value	Colors
0	gray
1	666
2	884

`-q n` Select shading options.

Flag	Option
1	shadows
2	reflectivity
4	refraction
8	highlighting on both sides of a surface
16	check if object casts shadows
32	primary (eyes) rays will be checked

To use, simply add the number of the features you want, for example, *shadows* + *reflectivity* = *1* + *2* = *3*. The parameter usage would be -q 3.

Render Flag	Default
-r 0	-q 63 (all ON)
-r 1	-q 8
-r 2	N/A

`-x` Set image x resolution.

Example: -x 640

`-y` Set image y resolution.

Example: -y 480

continues

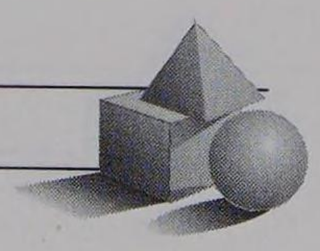
Table 2.1. Continued.

Parameter	Action
Miscellaneous	
`-Q`	Abort run by pressing any key.
`-R`	Resume an interrupted trace.
`-t n`	Display tracing information.
Miscellaneous	
	Value **Status** 0 none 1 totals at the end of trace 2 line 3 pixel (slows program down)
`-v`	Ray trace image from bottom to top.
`-V m`	Screen preview status. **Value** **Mode** 0 none 1 VGA
`-W`	Wait for a keypress to clear the screen.
`-z n`	Start ray tracing at line n. This feature is good for checking partial images.

A good approach to using command-line parameters is to create a series of batch files. Each batch file would have a different set of parameter combinations. You could use one of them to create a quick rough draft of the image and use another to create a final high-quality image.

Creating a Scene File

Creating a scene file is easy. In the following section, you start by using the text editor of your choice to create your first scene file. This section also helps you become familiar with Polyray's syntax.



Using a Text Editor

To create a scene file, use your favorite text editor, for example the EDIT editor included with MS-DOS 5.0 and 6.0. The text editor must be capable of saving the file in straight ASCII format. When you begin typing the scene file presented in one of the next sections, remember to use the Tab key to create multiple indentation levels in the file. This makes the file more readable.

Syntax Considerations

Polyray is case-sensitive, and you must type all keywords correctly for Polyray to parse the command. You don't need any programming skills to use Polyray; however, if you have programmed in C, it shouldn't take you long to get used to Polyray's syntax, because it has a C-language "feel." As an example, the braces—{ and }—are used extensively to denote the beginning and end of a definition. In the C language, the same brackets are used for roughly the same purpose.

A Detailed Look at a Scene File

Listing 2.1 shows an example of a Polyray scene file. You should run your text editor at this point and type the complete listing. This will help you become familiar with some of Polyray's keywords and format style. After typing the listing, read the following section, in which you'll "dissect" the scene file and study it in detail.

Save the file as DEMO.PI (this and all other listings are available in the companion disk).

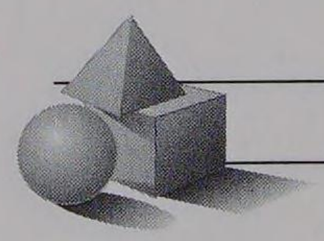
NOTE: All of the images in this book are set up for 100 × 100 resolution to minimize the rendering time. If you want to see more detailed, higher-resolution images on a 640 × 480 display, just change the resolution statement to

```
resolution      640,480
```

and the aspect statement to

```
aspect          1.33
```

Remember, however, that these higher-resolution images take much longer to render.



Listing 2.1. A first look at Polyray.

```
//DEMO.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <0.2, 1.0, -1.7>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
 }

// LIGHT_SOURCE
    light <0, 5, -10>

// PLANE (Polygon)
object {
   polygon 4, <-100,0,100>, <-100,0,-100>, <100,0,-100>, <100,0,100>
   translate <0, -1, 0>
   texture { checker matte_white, matte_black }
}

// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <0.5, 0.5, 0.5>
    shiny_green
}

// CONE
object {
    cone <0.0, -1, 0.0>, 0.5, <0.0, 0.0, 0.0>, 0.0
    translate <1.6, 0.48, 0>
    shiny_blue
}

// SPHERE
object {
    sphere < -0.96, 0, -1.12>, 0.25
    shiny_red
}
```

Header Section

Listing 2.2 contains the header section. In the first line, the symbol // is used to indicate a comment. Polyray ignores this line, but you will use it to identify the file. If you need more than one line of comments, simply add two slash marks at the beginning of each line you want to make a comment, as in the following:

```
// This is the first line
// This is another commented line
```

Listing 2.2. The header section of DEMO.PI.

```
//DEMO.PI

include "colors.inc"
include "texture.inc"
```

The two lines following the comment are very important. The `include` statement indicates that there is some information—in this case, color and texture information—that was defined in external files. In this case, the color definitions can be read from the file `colors.inc`, whereas texture definitions can be read from the file named `texture.inc`. You can have as many external files as necessary. This feature enables you to include as many files as needed which may have colors, textures, objects, or groups of objects. This eliminates the need for retyping the definitions each time you have to use them. You should take some time and become familiar with the `colors.inc` and `texture.inc` files.

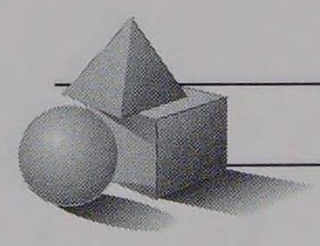
As we progress with the examples, you will need to save objects to a file for repeated access.

Viewpoint Definition Section

As mentioned in the first chapter, the `viewpoint` definition indicates the position of the observer and at which point the observer is looking. In this case (see Listing 2.3), the observer is positioned at X=0.2, Y=1.0, and Z=–1.7. Recall that we are using a left-handed coordinate system; therefore, the z axis points into the computer screen, whereas the x axis points to the right and the y axis points up. The observer is also looking at the origin <0.0, 0.0, 0.0 > (`at`). To orient the observer, the up direction is defined using the statement `up < 0.0, 1.0, 0.0 >`.

Listing 2.3. The viewpoint definition section of DEMO.PI.

```
viewpoint {
 from       <0.2, 1.0, -1.7>
 at         <0.0, 0.0, 0.0>
 up         <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
 }
```



The `resolution` statement is used to define the size of the final image: 100 × 100 pixels. The `aspect` ratio is the ratio of the width to the height: 100 / 100 = 1.0 (assuming the pixels are square). If an image size of 640 × 480 is used, the aspect ratio is 640 / 480 = 1.33. After the image is generated, the sphere will appear deformed. This is due to its closeness to the screen. A normal check for correct aspect ratio is, for example, that a sphere must appear to be round, unless you want to create a different effect. You might have to modify the aspect ratio value if you are using a non-square pixel video mode.

Light Source Definition

At least one light source has to be included in a scene file or your final image will be dark. To include a light source, use a definition similar to that in Listing 2.4. The only required parameters are the X, Y, and Z coordinate positions of the light source. If no color is attached to the definition, white light is assumed.

Listing 2.4. The light source definition of DEMO.PI.

```
// LIGHT_SOURCE
    light <0, 5, -10>
```

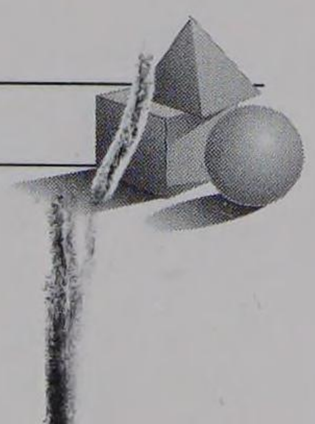
There are other types of light sources; however, this is the simplest example that can be presented. In the next chapter, you experiment with different light sources.

Object Definition

Listing 2.1 has four objects: a plane or polygon, a box, a cone, and a sphere. The following paragraphs describe each individually.

Plane

Polygons are used as planes to create a floor effect in a scene. You can construct a plane with a collection of four 3-D points to define the corners of the floor, as in Listing 2.5. The idea is to separate these four points as much as possible to create the illusion of an infinite floor.



Listing 2.5. Plane definition.

```
// PLANE (Polygon)
object {
   polygon 4, <-100,0,100>, <-100,0,-100>, <100,0,-100>, <100,0,100>
   translate <0, -1, 0>
   texture { checker matte_white, matte_black }
}
```

To lower the floor by one unit, the statement `<0, -1, 0>` is used to displace the points in the y axis. This is similar to the definition

```
polygon 4, <-100,-1,100>, <-100,-1,-100>, <100,-1,-100>, <100,-1,100>
```

Translating a plane down by a certain amount (as in Y = –1) creates the illusion that the objects on top of the plane are floating. Figure 4.16 in Chapter 4 shows three spheres and a plane which simulates water. Two of the spheres appear to be floating on top of the water, and the center sphere is in it.

Finally, the plane will have a checkered pattern consisting of `matte_white` and `matte_black` tiles as defined by `texture { checker matte_white, matte_black }`. More on this later (Chapter 4 deals entirely with light, colors, and textures).

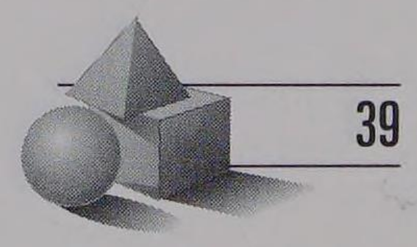
Box

The box object is initially a unit box with two opposing diagonal corners at <–1, –1, –1> (lower corner) and <1, 1, 1> (upper corner). To scale it down by half, you can use the `scale <0.5, 0.5, 0.5>` statement. This is similar to the definition `box <-0.5, -0.5, -0.5>, <0.5, 0.5, 0.5>` (see Listing 2.6).

At this point there is no benefit to using the box definition by itself or with the scaling operation, but it gives you the idea that objects can be manipulated by means of other operations.

Listing 2.6. Box definition.

```
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <0.5, 0.5, 0.5>
    shiny_green
}
```



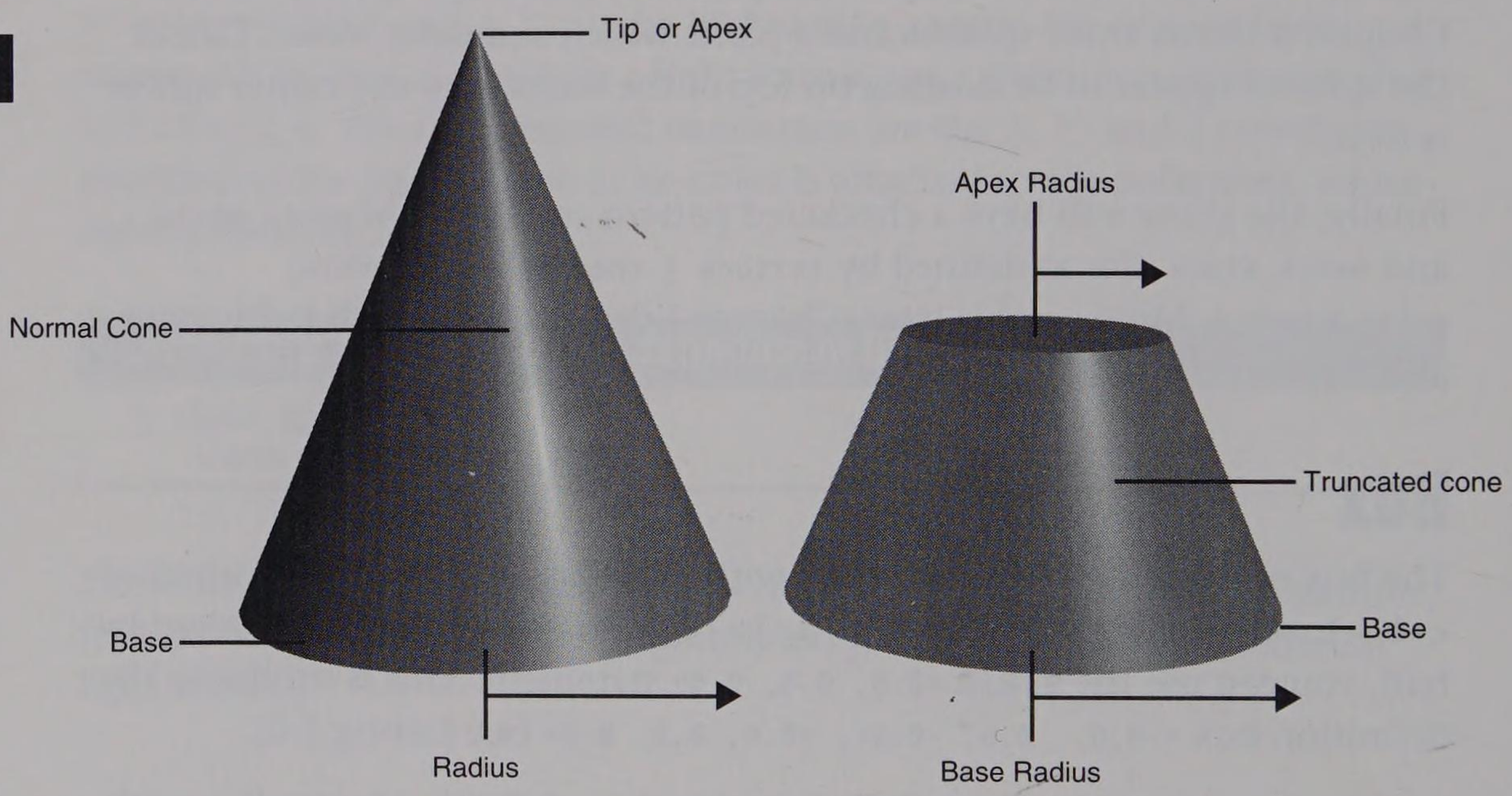
The surface of the box will have a `shiny_green` finish. Surface finishes are described later in this chapter in the "Textures" section.

Cone

Cones are described by the location of the center of the base and apex points. Also, the base and apex radii are required. In Listing 2.7, the base of the cone is located at <0.0, –1, 0.0> with a radius of 0.5 units, whereas the apex is located at <0.0, 0.0, 0.0> with a radius of 0.0. If the radius of the apex is different from zero, you will be defining a truncated cone (see fig. 2.1). To locate the cone in the final position, use the `translate <1.6, 0.48, 0>` statement.

Figure 2.1

Normal and truncated cones.



The final line, `shiny_blue`, attaches a surface color/finish to the cone.

Listing 2.7. Cone definition.

```
// CONE
object {
    cone <0.0, -1, 0.0>, 0.5, <0.0, 0.0, 0.0>, 0.0
    translate <1.6, 0.48, 0>
    shiny_blue
}
```

Sphere

The last object is the sphere. The sphere requires the location of its center point and its radius. In Listing 2.8, the center point is located at <–0.96, 0, –1.12> with a radius of 0.25.

The sphere has a `shiny_red` finish.

Listing 2.8. Sphere definition.

```
// SPHERE
object {
    sphere < -0.96, 0, -1.12>, 0.25
    shiny_red
}
```

Textures

In the scene file, notice that the plane object uses a texture keyword, whereas the other objects use just one word (for example, `shiny_green`). The reason behind this is that the words `shiny_red`, `shiny_green`, and `shiny_blue` have already been defined externally and were included in this scene by means of the `include "colors.inc"` statement. In other words, the `shiny_xxxx` words are basically just the name of a longer definition that was replaced with a short name.

To give a better example, let's replace the statement

```
texture { checker matte_white, matte_black }
```

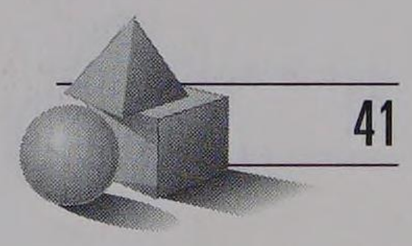
with a short name such as `CheckerWB`. You can do this with the keyword `define`, which you use in the following manner:

```
define   CheckerWB     texture { checker matte_white, matte_black }
```

Now all you need to do to call `texture { checker matte_white, matte_black }` is use the defined word CheckerWB, as in Listing 2.9.

Listing 2.9. Replacing a texture definition with a short name.

```
define   CheckerWB     texture { checker matte_white, matte_black }

// PLANE (Polygon)
object {
   polygon 4, <-100,0,100>, <-100,0,-100>, <100,0,-100>, <100,0,100>
   translate <0, -1, 0>
   CheckerWB       // this is the same as Listing 2.5
}
```



Textures usually require the declaration of various parameters, which makes the texture statement long. Therefore, it's a good idea to use the `define` keyword as much as possible to simplify the scene file, by replacing long definitions with a short name. You can either bundle them all in another file and include them in the header section or attach the definitions on the top of the scene file. You can use the `define` keyword to attach short names to anything. You explore this feature more in Chapter 3.

Ray Tracing a Scene File

If you typed Listing 2.1, you are almost ready to start ray tracing. The following sections explain how to simplify the process.

Creating a Batch File

As I mentioned in the "Command-Line Parameters" section, it is a good idea to create batch files to handle all the necessary command-line parameters. Using a text editor, create a batch file which contains the following line and name it DRAFT.BAT:

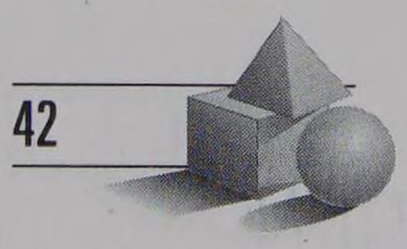
```
polyray %1.pi -r 0 -V 1 -t 0 -W -o %2.tga
```

Save the file in the same directory where Polyray resides. You'll use the DRAFT.BAT file to create a good-quality image without the time-consuming antialiasing methods. This means that some of the object edges might look rough. If you want a faster output, replace the `-r 0` parameter with either `-r 1` or `-r 2`. This reduces time, but it also lessens the final quality of the image.

Now, create a second batch file containing the following line and name it FINAL.BAT:

```
polyray %1.pi -r 0 -V 1 -t 0 -A -T 0.2 -S 25 -p 24 -W -o %2.tga
```

This file sets up Polyray to do antialiasing and saves the output with 24 bits per pixel. Use this batch file to create a high-quality final image. Needless to say, this method is very time-consuming, and you should use it after you have previewed the image using the DRAFT.BAT batch file.



Ray Tracing Your First Scene

Start by making the directory where Polyray resides the current directory. Type the following,

```
DRAFT  DEMO TEST1
```

at the DOS prompt and press the Enter or Return key. The input file is DEMO.PI and the output file will be TEST1.TGA. As the ray tracer generates the image line by line, you should be able to see a rough preview of the image. Because you are using the DRAFT.BAT file, the final image will be good, but not the best quality. This image is just a rough preview of the true output, so don't be disappointed.

A note about resolution: Most images in the book use the command "resolution 100,100" and "aspect 1.0". This is done to generate a quick image. You can modify these two statements in order to increase the size of the image and the aspect ration. Needless to say, the larger the image, the longer it takes to generate it; the smaller the image, the less quality you will see.

After the image is completed, press any key to go to DOS.

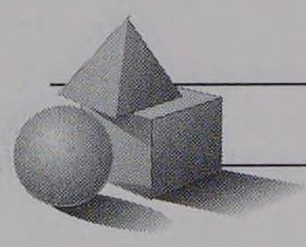
The next step is to run the final high-quality image by typing `FINAL DEMO TEST2` at the DOS prompt and pressing Enter. This image might take a little longer to generate than the previous image.

Both images will be of the same resolution, but the one generated with FINAL.BAT will be smoother and will offer more details (it also tends to be larger in size).

The next step is to use an image file viewer program to check the results of the ray tracer.

Viewing a Ray-Traced Image

After running the ray tracer, you have two image files: test1.tga and test2.tga. Both are compressed Targa files. Use your favorite image-file viewer, or the CompuShow viewer included on the bundled disk, and take a look at both files. Beware that some viewers do a poor job of showing image files. Also, the graphics card you have on your computer might have some effect on the colors that you see. It is highly recommended that you have a Super VGA card. Figures 2.2 and 2.3 display the output of files test1.tga and test2.tga respectively.



Both figures 2.2 and 2.3 show the same image, but with a subtle difference. Check for ragged lines on each, and look at the finer details.

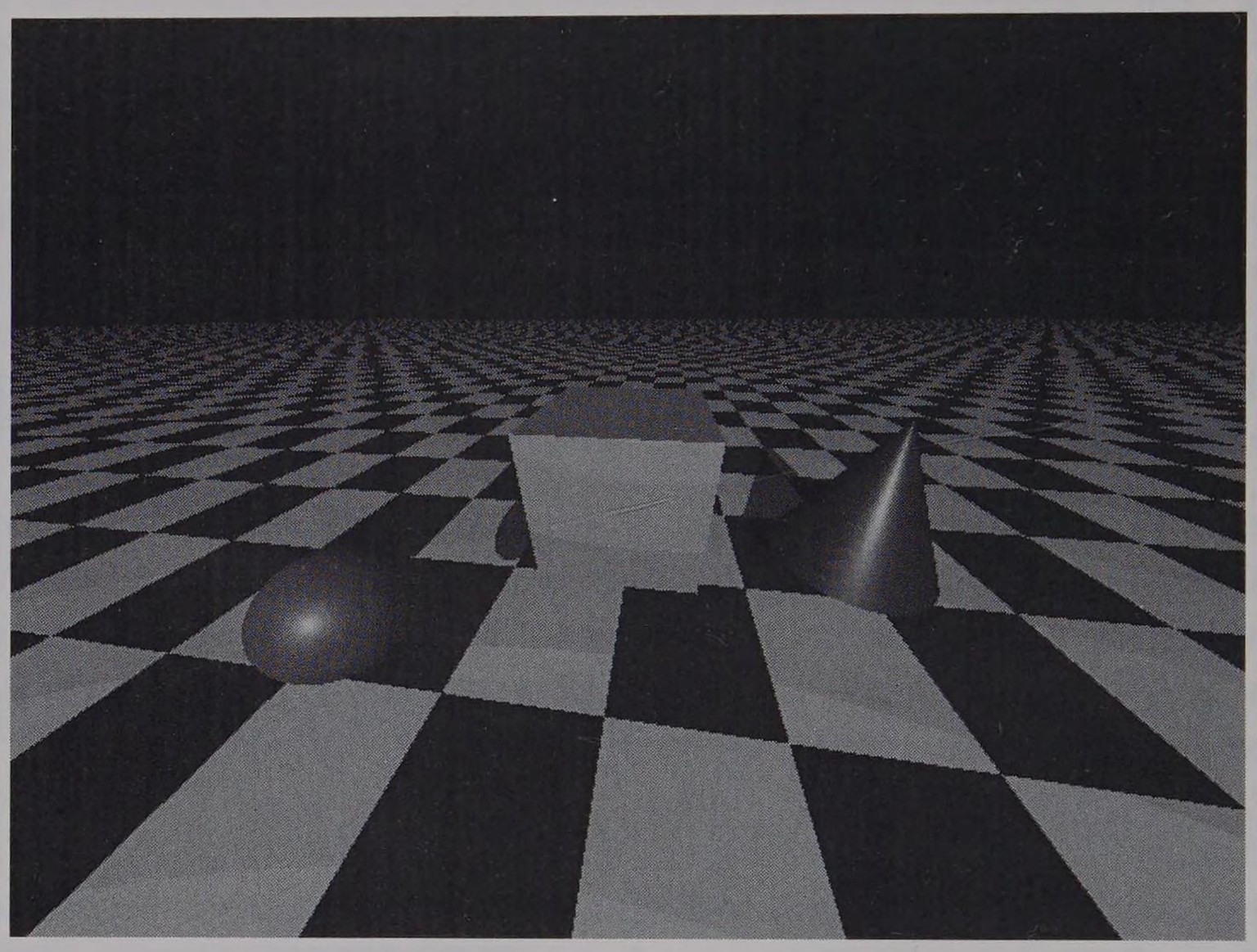


Figure 2.2

The output of test1.tga at 640 X 480 (aspect = 1.33).

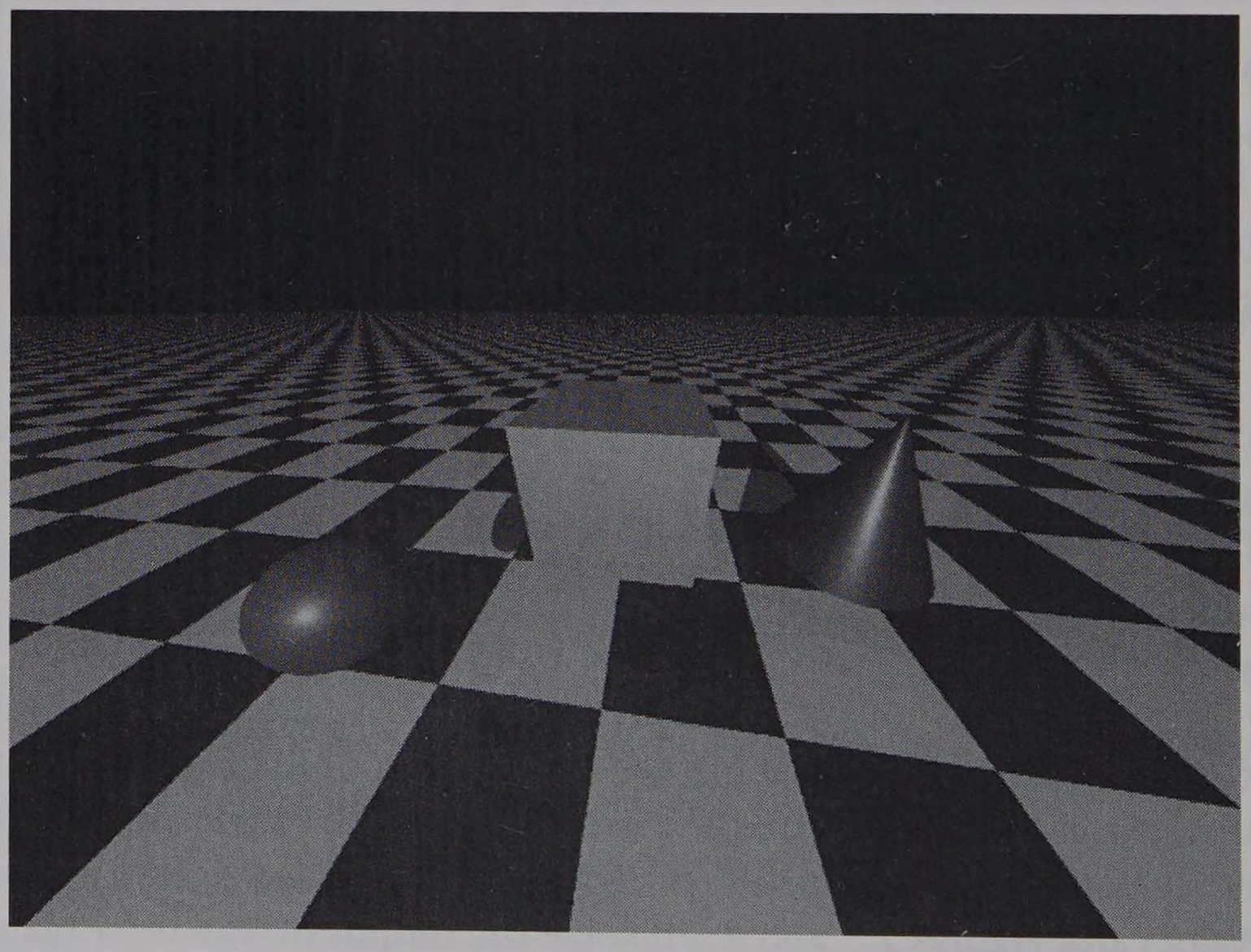


Figure 2.3

The output of test2.tga at 640 X 480 (aspect 1.33).

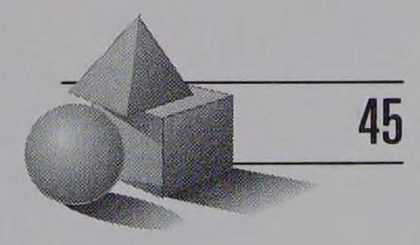
Summary

In this chapter, you were introduced to Polyray, an excellent ray tracer which helps you create photo-realistic images. Next, you saw the command-line parameters that you can use with Polyray. You use some of the parameters to enhance the quality of the final image by using antialiasing techniques and by increasing the number of bits per pixel. Use others parameters are used to control file output operations or miscellaneous features such as waiting for a key response at the end of the trace.

You also saw a sample scene file that was divided into five main sections: header, viewpoint, light source, object, and textures.

After reviewing the sample scene file, you received instructions on how to create batch files to simplify the ray tracing process. You created two images —in draft and final form—and received information on how to view them.

In the next chapter, you learn the Polyray ray-tracing program and learn how to create photo-realistic images in more detail. You will begin by creating simple objects, adding other objects to make complex objects, and finally, applying more of the characteristics of 3-D modelling by adding textured surfaces to the objects.

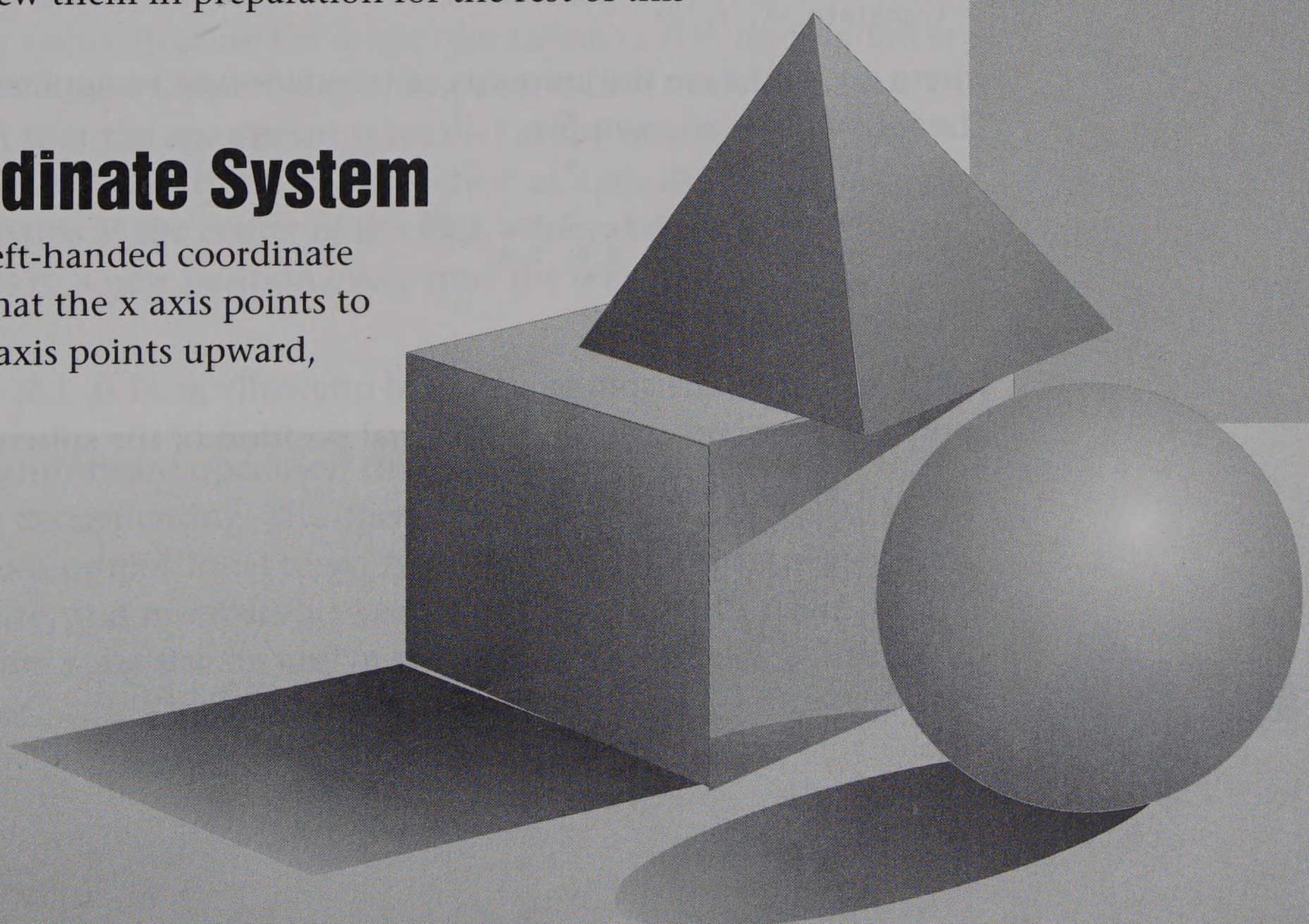


Using Polyray

The Polyray ray tracer is rich in features. In this chapter, we will expand on most of the concepts presented in Chapter 1. You have the opportunity to ray trace the available objects and experiment with different surfaces and strange objects, such as blobs. The chapter takes a detailed look at each object and includes sample scenes.

Background

The following four sections present the basic concepts discussed in Chapter 1 as they apply to Polyray. I suggest you take a moment to review them in preparation for the rest of this chapter.



3-D Coordinate System

Polyray uses a left-handed coordinate system. Recall that the x axis points to the right, the y axis points upward,

and the z axis points into the screen (refer to fig. 1.1). The coordinate system does not have any specific units. You can express the object dimensions in inches, feet, miles, or any other type of units. The important point to remember is to be consistent.

Vectors

Vectors are written as <X, Y, Z>, where X, Y, and Z are the x, y, and z coordinate values, respectively. Notice that the values are separated by commas. They are used to represent 3-D points such as the center of a sphere. Also, they can be used to represent a direction—as the vector <0, 1, 0> represents a direction along the positive y axis and the vector <–1, 0, 0> represents a direction along the negative x axis.

Transformations

Polyray supports many types of transformations. The most commonly used ones are translation, rotation, and scaling. These operations are used extensively throughout this book.

Translate

The translate operation moves the position of an object from the current position to a new position as determined by a vector. The syntax is

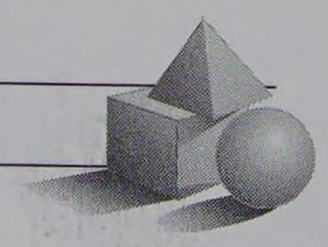
```
translate <X, Y, Z>
```

where X, Y, and Z are the amounts of translation to be applied to an object. The following is an example:

```
object {
     sphere <1.0, 1.5, -3.0>, 0.25
     translate <5.0, 6.0, 7.0>
     shiny_red
}
```

In this example, the sphere is located originally at <1.0, 1.5, –3.0>. After you apply the translate operation, the final position of the sphere is

```
X =  1.0 + 5.0 = 6.0
Y =  1.5 + 6.0 = 7.5
Z = -3.0 + 7.0 = 4.0
```



In other words, you can replace the previous example with the following:

```
object {
     sphere <6.0, 7.5, 4.0>, 0.25
     shiny_red
}
```

and you would get the same results.

Rotate

Use rotate <rx, ry, rz> to revolve an object about each axis. The object is rotated rx degrees about x, then ry degrees about y, and finally rz degrees about z. The rotations are in the same order they were described. Recall that the rotations follow the "left-hand rule" explained in Chapter 1 (refer to fig. 1.3 and fig. 1.4). The following is an example:

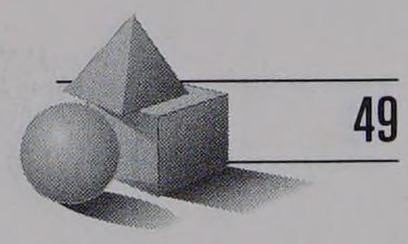
```
object {
     cone <0.0, -1.0, 0.0>, 0.5, <0.0, 0.0, 0.0>, 0.0
     rotate <90.0, 90.0, 0.0>
     shiny_red
}
```

Let's study the operations by steps. The first transformation is to rotate the cone by 90 degrees about x. Initially, the base of the cone is located at <0.0, –1.0, 0.0> and the apex is at <0.0, 0.0, 0.0>. After rotating 90 degrees about x, the base point is at <0.0, 0.0, –1.0> and the apex is still at <0.0, 0.0, 0.0>. In other words, the cone is aligned with the z axis.

Next, the rotation of 90 degrees about y is performed. The base is at <–1.0, 0.0, 0.0> and the apex is at <0.0, 0.0, 0.0>. In this case, the cone is now aligned with the x axis. Because the angle of rotation rz is 0, no rotation is applied about the z axis. Observe that the apex location never changed. This is due to the fact that the apex point is located at the origin, which is the center of rotation. You can think of a car wheel as a similar case. The center of the wheel remains at the center of the axle, whereas any other point on the wheel rotates to a new position every time the wheel turns.

Scale

Scaling is a straightforward operation that simply involves size incrementing or decrementing. The operation scale <sx, sy, sz> scales the object on each axis proportional to sx, sy, and sz. Any number greater than 1 increases the size, and numbers between 0 and 1 shrink the object. To keep the object the same size on any of the axes, use a value of 1. Using 0 gives you an error.



Viewpoint

In Chapter 1, you learned about the importance of positioning an observer, commonly called the "viewpoint" of the scene. In Chapter 2, we looked at an example of a scene file and the format used by Polyray to define such an observer. Polyray uses 13 parameters to define the viewpoint. Most of them are normally not used unless you want to experiment with effects such as changing the field of view or dissolving an image. The viewpoint format is shown in Listing 3.1.

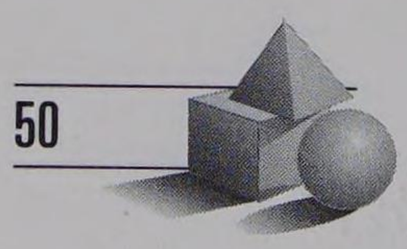
Listing 3.1. Viewpoint definition.

```
viewpoint {
      from < xo, yo, zo>
      at <xa, ya, za>
      up <xup, yup, zup>
      resolution xres, yres
      aspect    ar
      angle fov
      max_trace_depth n
      hither d
      yon d
      dither_rays value
      dither_objects value
      aperture a
      focal_distance d
}
```

The description of each parameter is presented in Table 3.1.

Table 3.1. Viewpoint parameters.

Parameter	Meaning
Commonly Used Parameters	
`from`	Observer is located at <xo, yo, zo>; default vector is <0, 0, –1>.
`at`	Observer is looking at point <xa, ya, za>; default vector is <0, 0, 0>.
`up`	The "up" direction relative to the observer; default vector is <0, 1, 0>. **Examples:** <0, 1, 0> = along y axis. <1, 0, 0> = along x axis. <1, 1, 0> = along a direction 45 degrees from the x axis.



`resolution`	Final image size; this size is independent of the screen resolution; default values 512, 512. **Example:** resolution 640, 480 means xres = 640, yres = 480
`aspect`	If the screen pixels are square, then ar = xres/yres; default value = 1. **Example:** ar = 640/480 = 1.33
`angle`	Indicates the field of view in degrees, from top to bottom of the image; default = 45 degrees.
`max_trace_depth`	Scenes with high values of reflection or transparency require more calculations; the number "n" determines the maximum number of levels deep of recursion that Polyray will compute; default = 5. **Example:** `max_trace_depth 3`

Special-purpose Parameters

`hither`	Sets a lower limit on intersections; anything closer to the viewpoint than "d" is ignored, default = 1e–3.
`yon`	Sets an upper limit on intersections; anything farther from the viewpoint than "d" is ignored, default = 1e5.
`dither_rays`	Polyray generates an internal random number; If this number is greater than value, the current light ray is skipped; value is between 0 and 1.
`dither_objects`	Same as dither_rays, but the current object is skipped.
`aperture`	Used to produce depth of field blurred images; must be greater than 0 to produce blur; Values between 0.1 and 0.6 are suggested.
`focal_distance`	Distance from observer to the point being focused at; the value d defaults to the distance between from and at. This value is only relevant if aperture is greater than zero.

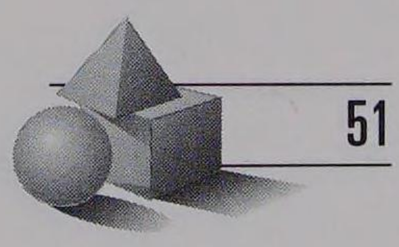


Table 3.1 divides the viewpoint parameters into two groups: commonly used and special purpose. Normally, the commonly used parameters appear in the scenes. Most of these parameters have default values; however, you'll need to modify them to suit your needs. A common viewpoint definition is presented in Listing 3.2. The variables `xa`, `ya`, `za`, `xo`, `yo`, and `zo` must be replaced with values.

Listing 3.2. Standard viewpoint definition.

```
viewpoint {
     from <xo, yo, zo>
     at <xa, ya, za>
     up <0, 1, 0>
     resolution 640, 480
     aspect 1.33
}
```

The examples discussed in this book use a resolution of (100, 100) and an aspect ratio of 1. This is done so that you can preview the final images quickly. The images displayed throughout the book are the same ones as the listing examples but the resolution was changed to (640, 480) with an aspect ratio of 1.33 to increase the quality. Some of the images took as long as four hours to complete because the Jitter antialiasing method was used to enhance the quality.

All the samples have their own viewpoint definition, which you can modify to view other areas of the image.

Naming Definitions

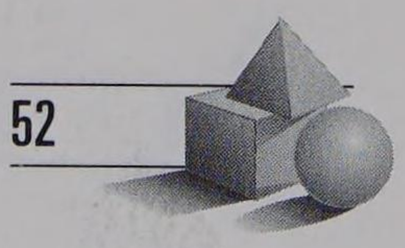
Polyray uses the special keyword `define` to attach token names to standard definitions. The syntax for `define` is

```
define NAME expression
define NAME transform {.....}
define NAME object {.....}
define NAME texture {.....}
define NAME surface {.....}
```

where `NAME` is the name of the object or expression that is being defined on the right side of the `define` statement.

For example, to attach a name to a sphere, use the following:

```
define MySphere object {
                    sphere <0,0,0>, 1
                }
```



You can use the following to position the sphere elsewhere and attach a color to it:

```
object {
     MySphere
     translate <1, 3.4, 5.2>
     shiny_red
}
```

As you can see, the entire expression is not substituted but the object that's being defined is:

```
object {
     sphere <0,0,0>, 1
     translate <1, 3.4, 5.2>
     shiny_red
     }
```

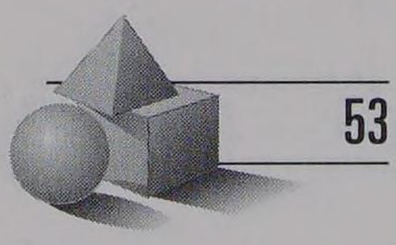
Throughout this book, you'll see examples of the use of the `define` keyword.

Creating Objects

Polyray has a wide variety of objects which are referred to as *primitives*, such as the sphere, cone, cylinder, and others. In Chapter 1, you read about modeling common everyday objects by starting with simpler primitives. In the following sections, each object is described in detail with examples of their use. Due to their complexity, some objects—for example, Bézier patches, lathe objects, and extruded objects—require external programs or utilities to create them. The disk that comes with this book includes a modeler called POVCAD, which can help you easily create lathe and extruded surfaces. In fact, most of the scenes included in this book were initially created in POVCAD and then exported to Polyray. Chapter 7, "Using a 3-D Modeler," describes POVCAD in more detail.

Creating Simple Objects

Polyray primitives are divided into three groups: simple objects, modeling surfaces, and miscellaneous objects. Simple objects are usually defined with a few constants and location vectors, for example, the sphere. The sphere needs a vector for the location of the center and a radius. Modeling surfaces are either functions of the type f = (x,y,z), collections of triangles that make an object, or Bézier patches that require 16 points to define the whole surface. Finally, miscellaneous objects are those that cannot be categorized



in any of the previous two groups but still are interesting and useful, for example, one of them can be used to create landscapes. The following sections describe some simple objects.

Box

The box object can be declared by two corner points. The syntax is

```
box <xl, yl, zl>, <xu, yu, zu>
```

in which `xl`, `yl`, `zl` are the coordinates of the front lower left corner, and `xu`, `yu`, `zu` are the coordinates of the back upper right corner

Listing 3.3 shows an example of using the box primitive, and figure 3.1 displays the final image.

Listing 3.3. Using the box primitive (BOXES.PI).

```
//BOXES.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <-0.5, 2.0, -5.0>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 resolution      100,100

 aspect          1.0
}

// LIGHT_SOURCE
    light <0.0,  0.0, -3.0>
// LIGHT_SOURCE
    light <0.5,  3.0, -3.0>
// LIGHT_SOURCE
    light <1.0,  0.0, -5.0>

// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    reflective_yellow
}

// BOX
```

```
object {
    box <-1, -1, -1>, <1, 1, 1>
    translate <-1.65, -2.1, 0>
    white_marble
}

// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <2, 1, 3>
    translate <2.4, -2.1, 0>
    dented_red
}
```

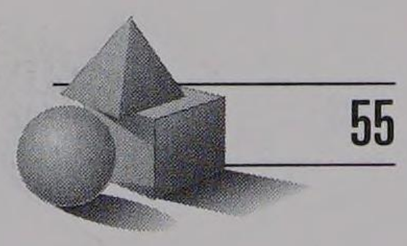
Observe how only one type of box was defined, `box <-1, -1, -1>, <1, 1, 1>`. Using this box is practical because it's easy to size just by applying a scaling factor. The box is normally called a *unit box*. In other words, the first two boxes are unit boxes and the third one was scaled to <2, 1, 3> in the x, y, and z axes, respectively. The third box could have been written as `box <-2, -1, -3>, <2, 1, 3>`.

Sometimes it's much easier to define a standard object such as the unit box and then scale it appropriately than it is to modify all the parameters, which is an error-prone operation. For example, use the `define` keyword to create a unit box object and modify the last box in Listing 3.4.

Listing 3.4. Using the define keyword to create a unit box.

```
define UnitBox  object {
                    box <-1, -1, -1>, <1, 1, 1>
                    }

object {
     UnitBox
     scale <2, 1, 3>
     translate <2.4, -2.1, 0>
     dented_red
}
```

The power of the `define` keyword becomes more apparent when you deal with complex scene files.



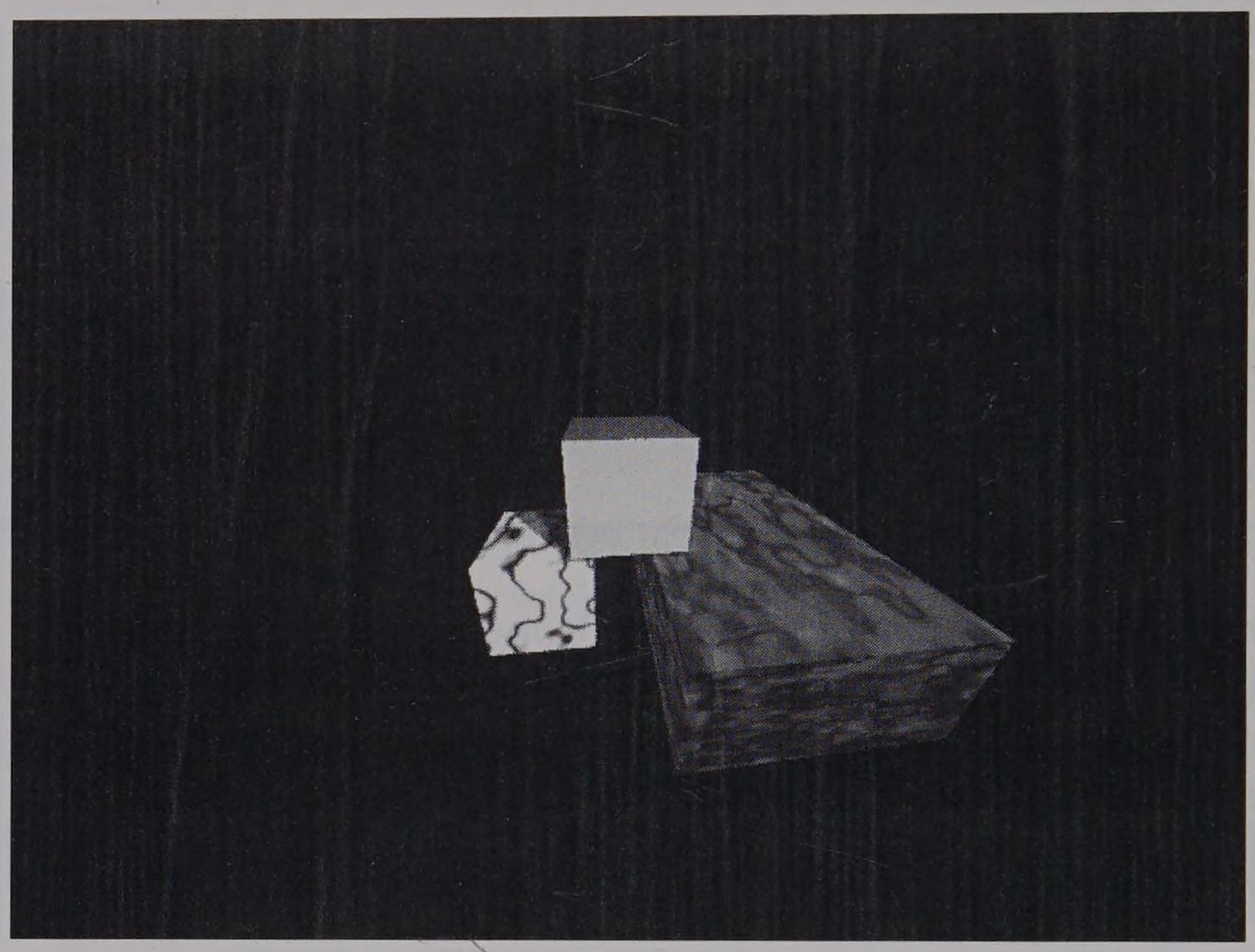


Figure 3.1

Box example (BOXES.PI).

Plane

If you recall Listing 2.1, we used a polygon definition to create a floor. Listing 3.5 shows the section of Listing 2.1 in which the plane was defined.

Listing 3.5. Plane definition extracted from Listing 2.1.

```
// PLANE (Polygon)
object {
   polygon 4, <-100,0,100>, <-100,0,-100>, <100,0,-100>, <100,0,100>
   translate <0, -1, 0>
   texture { checker matte_white, matte_black }
}
```

To create the effect of an infinite plane, the coordinates of the polygon were much higher numbers than the rest of the objects. It is useful to have a plane object to create floors or walls in your scene. Because Polyray does not have a plane object, we will create three, each one perpendicular to an axis.

The first step is to create an infinite plane. You can do this by using a definition such as

```
polygon 4,      <-10000,0,10000>, <-10000,0,-10000>,
                <10000,0,-10000>, <10000,0,10000>
```

This ensures that the polygon is always larger than the other objects.

Another way of doing this is to use a `define` statement such as

```
define UnitPlane object { polygon 4, <-1,0,1>, <-1,0,-1>,
}                                    <1,0,-1>, <1,0,1>
define InfinitePlane object { UnitPlane scale <10000, 1, 10000> }
```

In this example, the plane called `InfinitePlane` is extremely large in the x and z axes. It's a personal choice to use the previous method or

```
polygon 4,      <-10000,0,10000>, <-10000,0,-10000>,
                      <10000,0,-10000>, <10000,0,10000>
```

The next step is to define the planes depending on their orientation:

```
define Plane_XZ object {
                          polygon 4, <-10000,0,10000>,
                                                <-10000,0,-10000>,
                                                <10000,0,-10000>,
                                                <10000,0,10000>
                 }

define Plane_XY  object {
                          polygon 4, <-10000,10000,0>,
                                                <10000,10000,0>,
                                                <10000,-10000,0>,
                                                <-10000,-10000,0>
                          }

define Plane_XZ  object {
                          polygon 4, <-10000,0,10000>,
                                                <-10000,0,-10000>,
                                                <10000,0,-10000>,
                                                <10000,0,10000>
                          }
```

Listing 3.6 presents a scene in which three planes, a cone, and a sphere are used. The planes have mirror finishes, and the cone and the sphere are made of stone. The planes are located in a fashion similar to a room with two walls but no ceiling. The cone and the sphere are in the center of the room. The scene looks extremely "busy" due to the many reflections.

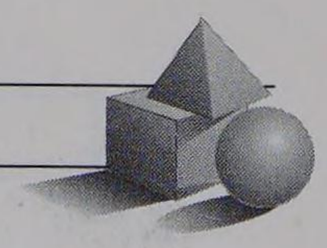
Listing 3.6. Defining and using planes (PLANES.PI).

```
//PLANES.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

define Plane_YZ  object {
                  polygon 4, <0,-10000,10000>, <0,10000,10000>,
                  <0,10000,-10000>, <0,-10000,-10000>
                 }
define Plane_XY  object {
```

continues

Listing 3.6. Continued.

```
                    polygon 4, <-10000,10000,0>, <10000,10000,0>,
                    <10000,-10000,0>, <-10000,-10000,0>
                   }

define Plane_XZ    object {
                    polygon 4, <-10000,0,10000>,  <-10000,0,-10000>,
                    <10000,0,-10000>, <10000,0,10000>
                   }

viewpoint {
 from       <0.0, 1.0, -4.0>
 at         <0.0, -0.25, 0.0>
 up         <0.0, 1.0, 0.0>
 angle           45
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
   light <5.0, 5.0, -5.0>

// PLANE (Polygon)
object {
// normal in X direction
    Plane_YZ
    translate <-0.6, 0, 0>
    mirror
}

// PLANE (Polygon)
object {
// normal in Z direction
    Plane_XY
    translate <0, 0, 0.6>
    mirror
}

// PLANE (Polygon)
object {
// normal in Y direction
    Plane_XZ
    translate <0, -0.6, 0>
    mirror
}

// SPHERE
object {
    sphere < 0, 0, 0>, 0.25
    Stone5
}

// CONE
object {
    cone <0.0, -0.5, 0.0>, 0.2, <0.0, 0.0, 0.0>, 0.0
    translate <0.8, 0.4, -0.8>
    Stone6
}
```



Observe that in the header section, a new include file named `stones.inc` was used. This file has the definition for the `Stone5` and `Stone6` textures used on the last two objects. The file `stones.inc` has approximately 23 different stone textures.

After the header section, three "infinite" planes were used, all of which have mirror finishes. Figure 3.2 shows the many reflections produced as a result of the mirrored finishes.

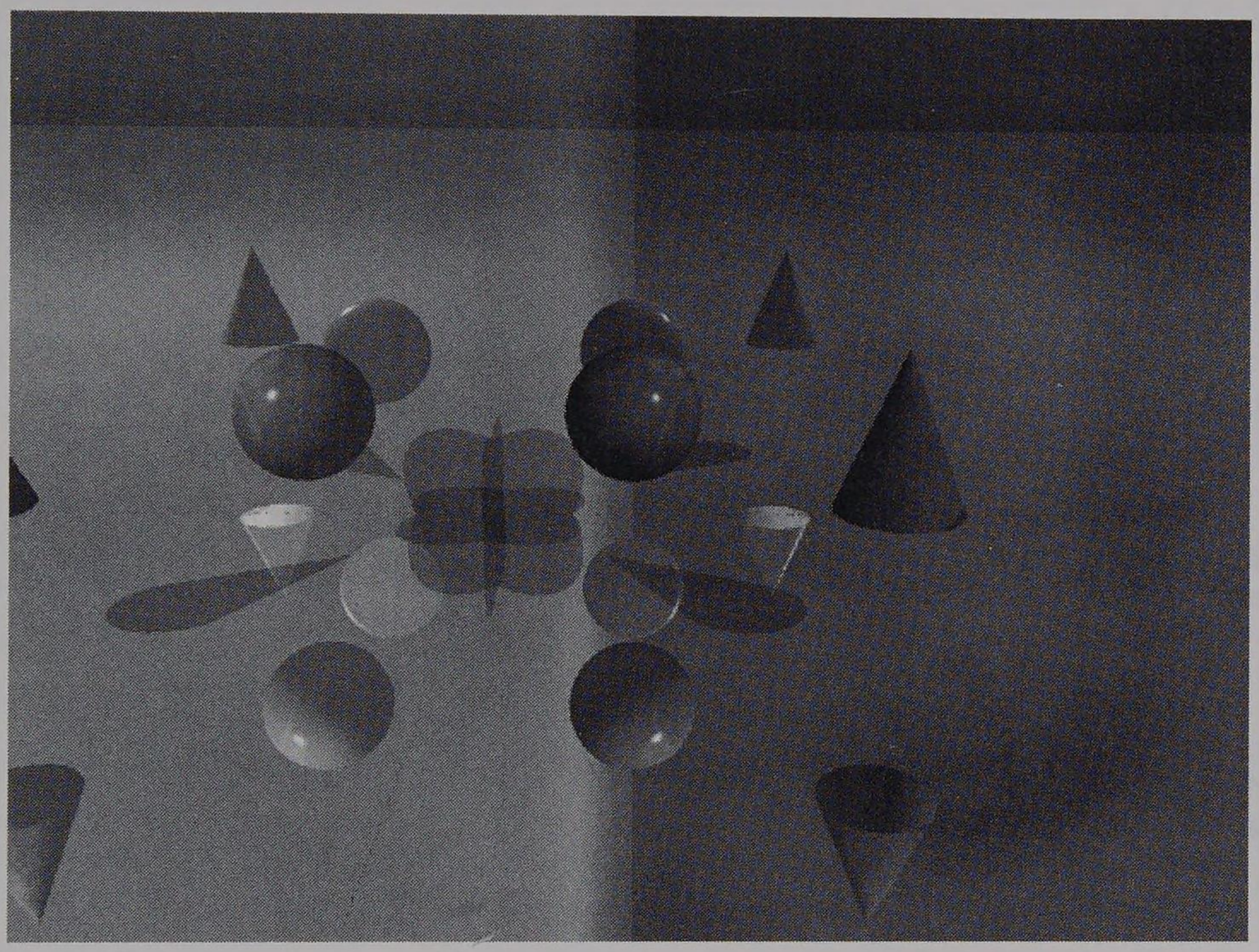


Figure 3.2
Using planes as mirrors (PLANES.PI).

Sphere

Spheres are the most common ray-traced object. Why? Because a sphere is the fastest ray-traceable object. A good application for spheres is to test various textures and colors. You can create an array of spheres, each with a different color or finish. You can then use the final image for comparison purposes. Various sphere examples have been presented in previous chapters. The syntax is as follows:

```
sphere <xc, yc, zc>, radius
```

where `xc`, `yc`, and `zc` define the center point of the sphere. Listing 3.7 shows a sphere example that generates the images shown in figure 3.3.

Listing 3.7. Using spheres (SPHERES1.PI).

```
//SPHERES1.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

viewpoint {
 from      <0.0, 0.0, -2.0>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
    light <0, 5, -5>

// LIGHT_SOURCE
    light <0, 10, 0>

// LIGHT_SOURCE
    light <5, 5, 10>


// SPHERE
object {
    sphere < 1.6, 0, 0>, 0.5
    Stone1
}

// SPHERE
object {
    sphere < 1.38564, 0.8, 0>, 0.5
    Stone2
}

// SPHERE
object {
    sphere < 0.8, 1.38564, 0>, 0.5
    Stone3
}

// SPHERE
object {
    sphere < -0, 1.6, 0>, 0.5
    Stone4
}

// SPHERE
object {
    sphere < -0.8, 1.38564, 0>, 0.5
    Stone5
```

```
}

// SPHERE
object {
    sphere < -1.38564, 0.8, 0>, 0.5
    Stone6
}

// SPHERE
object {
    sphere < -1.6, -0, 0>, 0.5
    Stone7
}

// SPHERE
object {
    sphere < -1.38564, -0.8, 0>, 0.5
    Stone8
}

// SPHERE
object {
    sphere < -0.8, -1.38564, 0>, 0.5
    Stone9
}

// SPHERE
object {
    sphere < 0, -1.6, 0>, 0.5
    Stone10
}

// SPHERE
object {
    sphere < 0.8, -1.38564, 0>, 0.5
    Stone11
}

// SPHERE
object {
    sphere < 1.38564, -0.8, 0>, 0.5
    Stone12
}

// SPHERE
object {
    sphere < 0, 0, 0>, 0.8
    Stone13
}
```

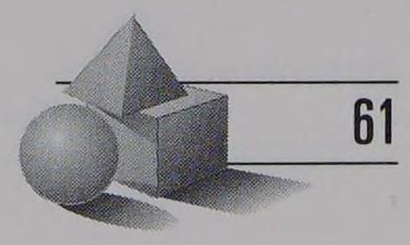
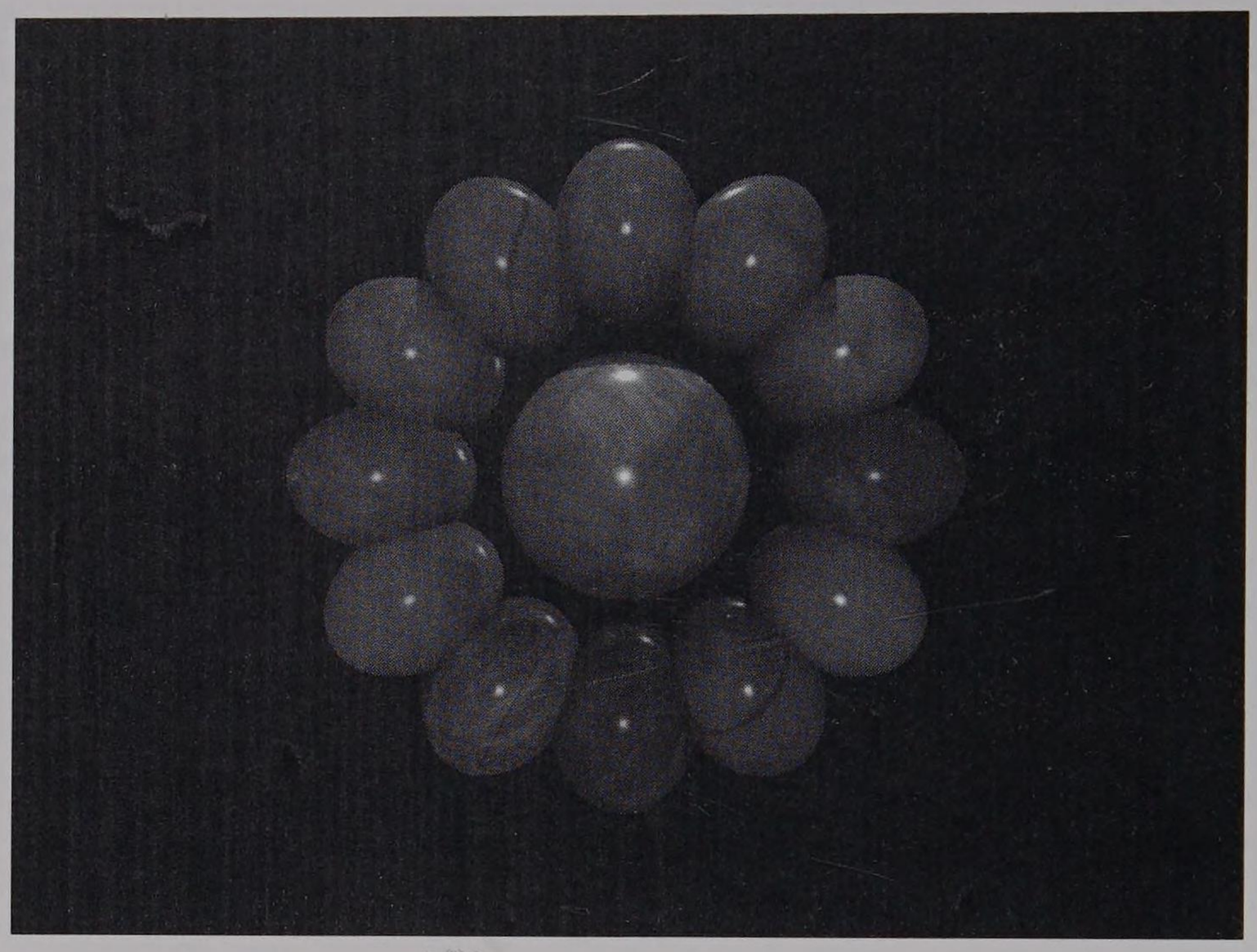


Figure 3.3

Using spheres (SPHERES1.PI).



In the scene file spheres1.pi, 12 spheres are positioned in a circle, each rotated about the z axis 30 degrees apart from the one before it. The equations to calculate each position are

X = 1.6 COS (angle) where angle = 0, 30, 60,...,330 degrees

Y = 1.6 SIN (angle)

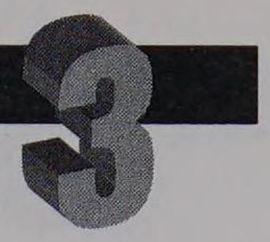
Z = 0.0

I didn't do the calculations myself—I used POVCAD to generate the spheres automatically! In Chapter 7, the section "Creating Multiple Copies of Objects" (Tutorial #3) explains how to create objects similar to these (see also figs. 7.9 and 7.10).

The last sphere is larger than the rest, and it is located at the origin. Observe that I used 13 types of stone textures to differentiate the spheres. This scene would require you to compute the X and Y positions of all the spheres (assuming you didn't have POVCAD). Listing 3.8 shows an easier way.

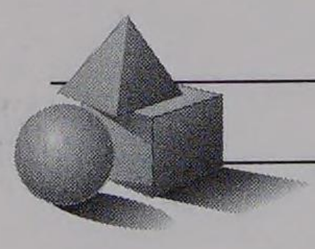
Listing 3.8. Simplifying Listing 3.7 (SPHERES2.PI).

```
//SPHERES2.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"
```



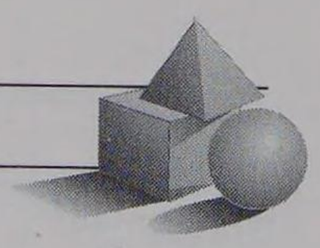
```
define FirstSphere object {
                      sphere < 1.6, 0, 0>, 0.5
                    }

viewpoint {
 from       <0.0, 0.0, -2.0>
 at         <0.0, 0.0, 0.0>
 up         <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
    light <0, 5, -5>

// LIGHT_SOURCE
    light <0, 10, 0>

// LIGHT_SOURCE
    light <5, 5, 10>


// SPHERE
object {
    FirstSphere
    Stone1
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 30>
    Stone2
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 60>
    Stone3
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 90>
    Stone4
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 120>
    Stone5
}

// SPHERE
```

continues



Listing 3.8. Continued.

```
object {
    FirstSphere
    rotate <0, 0, 150>
    Stone6
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 180>
    Stone7
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 210>
    Stone8
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 240>
    Stone9
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 270>
    Stone10
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 300>
    Stone11
}

// SPHERE
object {
    FirstSphere
    rotate <0, 0, 330>
    Stone12
}

// SPHERE
object {
    sphere < 0, 0, 0>, 0.8
    Stone13
}
```



Although this file is as long as the SPHERES1.PI file, it saves you from having to calculate the positions.

Both listings produce identical results in terms of the location of the spheres. The textures might vary due to rounding errors. The difference between SPHERE1.PI and SPHERE2.PI resides in the use of the rotate statement to move the original sphere to its new position. The statement rotate <0, 0, D>—where D varies from 0 to 330—is extremely useful, and you should keep it in mind for cases like this. The trick is to locate the initial sphere in a position outside the origin. Recall that if the object is at the origin, any rotation applied to it does not translate it.

After locating the sphere outside the origin, the `rotate` statement is applied, which has the same effect as a child spinning a rope with a stone tied to it. The hand of the child is the origin and the length of the rope is the original displacement of the sphere. The spin is the increase in rotation angle, from 0 to 330 degrees. Observe that the original sphere is being rotated at every step, creating a new sphere. Sometimes this is referred to as *orbital rotation* due to its resemblance to a satellite orbiting the Earth.

Disc

The disc object behaves as two different objects. It can be either a circular plate (thin coin) or a washer (coin with a hole in the middle). The syntax is as follows

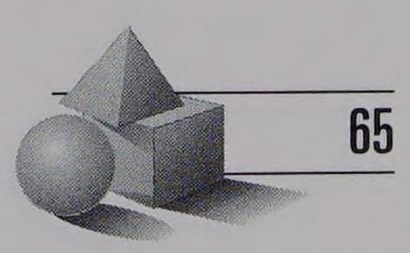
circular plate	`disc <xc, yc, zc>, <nx, ny, nz>, r`
washer	`disc <xc, yc, zc>, <nx, ny, nz>, ir, or`

in which `r` is the radius of the circular plate, `ir` is the inner radius of the washer, `or` is the outer radius of the washer, `<xc, yc, zc>` is the center of the object, and `<nx, ny, nz>` defines the normal vector of the disc. Think of the normal as a line that goes from `<xc, yc, zc>` to `<nx, ny, nz>` and is always perpendicular to the disc. You can imagine the disc as a wheel and the normal is an axle that goes through its center. The axle is always perpendicular to the wheel.

Example 1:

```
disc <0,0,0>, <0,1,0>, 0.5
```

This is a circular plate located at the origin, with the normal pointing along the y axis. The disc is in the xz plane.



Example 2:

```
disc <0,0,0>, <1,1,0>, 0.5
```

This is a disc, again at the origin, with the normal at 45 degrees relative to the x and y axis. If you look into the z axis, you see the disc slanted at 45 degrees.

An easier way to handle discs is to use `define` statements and create known positions that could be easily rotated to a new position (see Listing 3.9).

Listing 3.9. Defining standard disc positions.

```
define Disc_XZ object {disc <0,0,0>, <0,1,0>, 1}

define Disc_XY object {disc <0,0,0>, <0,0,1>, 1}

define Disc_YZ object {disc <0,0,0>, <1,0,0>, 1}
```

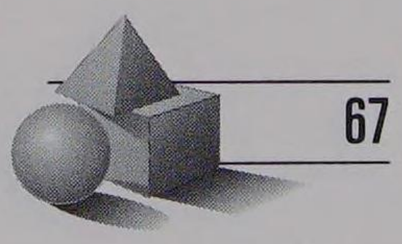
Listing 3.9 helps you simplify the use of discs. Remember to use the `scale` command to set the correct sizes, and the `rotate` and `translate` commands to move and rotate the disc.

The washer definition is more difficult to simplify than the disc because the values `ir` and `or` are required. Listing 3.10 shows an example of discs. Figure 3.4 shows the output.

Listing 3.10. Using discs (DISCS.PI).

```
//DISCS.PI

include "colors.inc"
include "texture.inc"

define Plane_XZ  object {
                        polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                        <10000,0,-10000>, <10000,0,10000>
                    }

define CheckerWB texture {checker matte_white, matte_black}
viewpoint {
 from      <0.0, 1.2, -5.0>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
    light <0, 0, -5>
```

```
// LIGHT_SOURCE
    light <1, 1, -5>


// PLANE (Checkered Floor)
object {
    Plane_XZ
    CheckerWB
    scale <2, 1, 1>   // make tiles larger!
    translate <0, -3, 0>
}

// SPHERE
object {
    sphere <0, 1, -1>, 0.7
    shiny_red
}

// WASHERS(Rings)
object {
    disc < 0, 0, 0>, <0, 1, 0>, 0.7, 0.9
    reflective_yellow
}

object {
    disc < 0, -0.2, 0>, <0, 1, 0>, 1.2, 1.5
    reflective_yellow

}

object {
    disc < 0, -0.4, 0>, <0, 1, 0>, 1.7, 1.9
    reflective_yellow
}

object {
    disc < 0, -0.6, 0>, <0, 1, 0>, 2.2, 2.4
    reflective_yellow
}

object {
    disc < 0, -0.8, 0>, <0, 1, 0>, 2.7, 2.9
    reflective_yellow
}

object {
    disc < 0, -1.0, 0>, <0, 1, 0>, 3.2, 3.4
    reflective_yellow
}

//DISC
object {
    disc <0, -5, 10>, <0, 0, 1>, 10.0
    rotate<-35, 0, 0>
    mirror
}
```



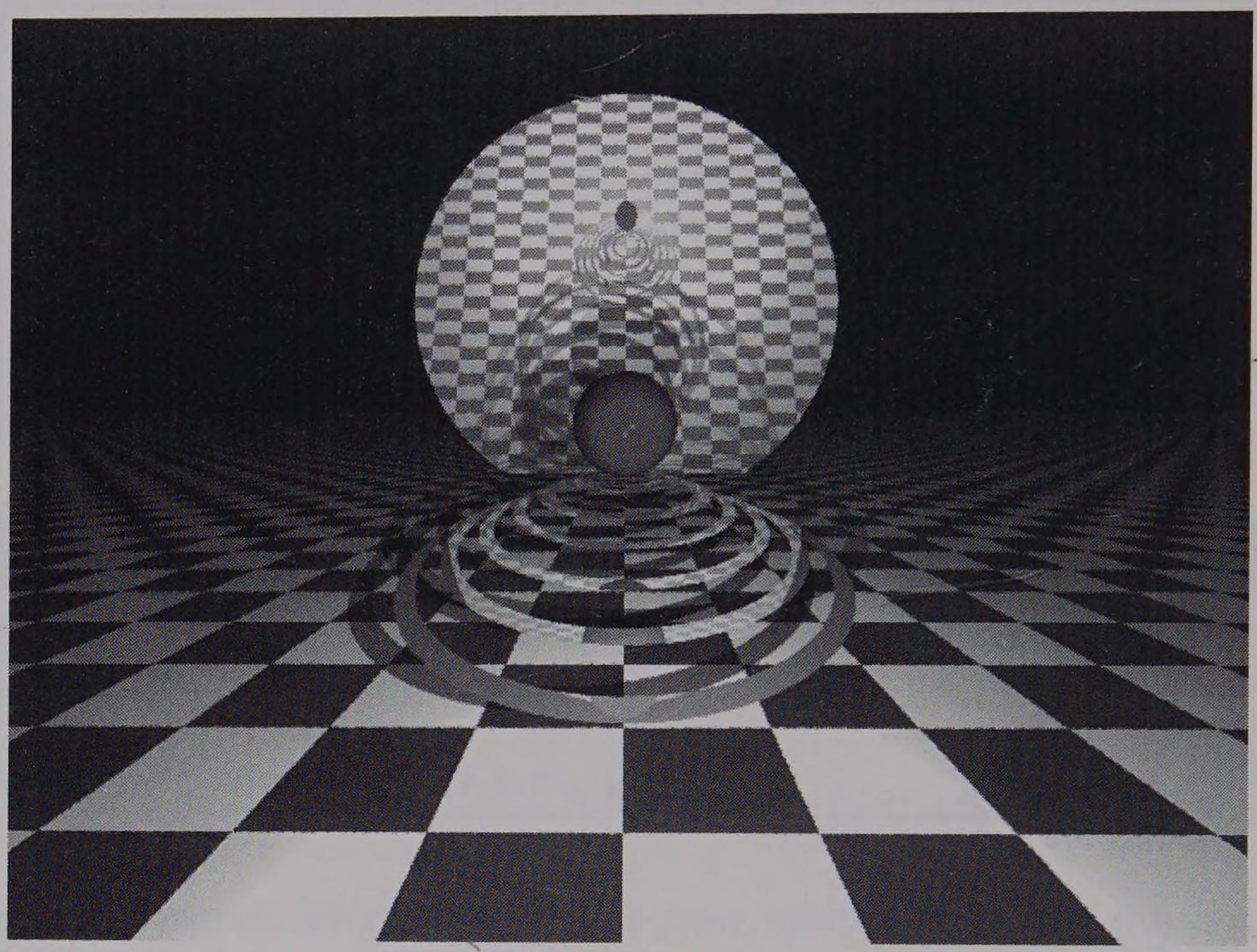


Figure 3.4

Creating discs and rings (DISCS.PI).

The scene DISCS.PI has some interesting features. Washers are positioned on top of each other along the Y axis, separated by a certain distance. There is a sphere on top of the rings. Observe how the sphere is reflected on the rings. At the back of the rings and sphere, a disc is placed to act as a mirror. It is slanted to reflect both the rings and the checkered floor. Note that the `Plane_XZ` and the `CheckerWB` definitions were included in the header section.

Parabola

Parabolas require an apex point (the tip of the curve), a base point, and the radius at the base point. The syntax is

```
parabola <xa, ya, za>, <xb, yb, zb>, r
```

in which `xa`, `ya`, and `za` are the coordinates of the apex, `xb`, `yb`, and `zb` are the coordinates of the base, and `r` is the radius of the base.

It is easier to define a parabola by aligning it with an axis and then rotating it to its required position. Listing 3.11 shows a quick example of parabolas (see fig. 3.5 for the output). Two parabolas are defined. One of them is rotated 180 degrees so that their bases meet.

Listing 3.11. Using parabolas (PARABOLA.PI).

```
//PARABOLA.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from       <0, 0, -5.0>
 at         <0.0, 0.0, 0.0>
 up         <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
    light <1, 6, -6>

// LIGHT_SOURCE
    light <0, -7, -6>

// PARABOLA
object {
        parabola <0, 5, 0>, <0, 0, 0>, 3
        white_marble
}


// PARABOLA
object {
        parabola <0, 5, 0>, <0, 0, 0>, 3
        rotate <0, 0, 180>
        sapphire_agate
}
```

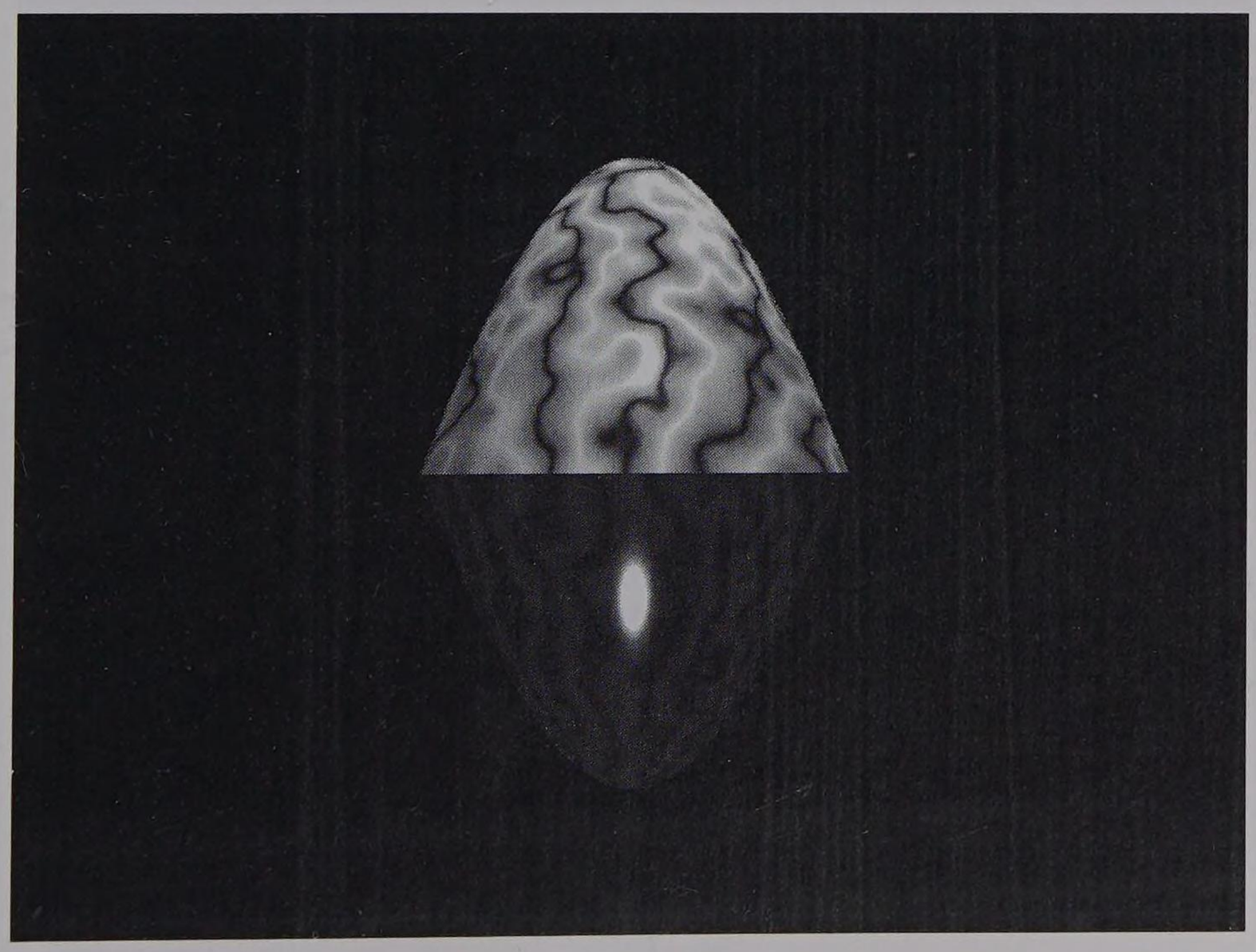


Figure 3.5

Two parabolas (PARABOLA.PI).

Parabolas are not closed (capped). If you need to close a parabola, use a disc object of the same size and orientation as the parabola's base.

Cone

Cones have four parameters: two points (base and apex) and two radii. The syntax is

```
cone <xb, yb, zb>, rb, <xa, ya, za>, ra
```

If one of the two radii, `ra` or `rb`, is equal to zero, the corresponding point becomes the apex of the cone. As with parabolas, cones are not closed. Refer to Listing 3.6 (PLANES.PI) for an example of using cones.

Cylinder

Cylinders require two endpoints and a radius:

```
cylinder <xo, yo, zo>, <x1, y1, z1>, r
```

Listing 3.12 shows an example of cylinders (see fig. 3.6 for the output).

Listing 3.12. Using cylinders (CYLINDER.PI).

```
//CYLINDER.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <0.1, 0.0, -0.7>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
   light <0, 5, -5>

// LIGHT_SOURCE
   light <10,1,0>

// CYLINDER
object {
    cylinder <0.0, -0.5, 0.0>, <0.0, 0.5, 0.0>, 0.2
    rotate <0, 0, 90>
    reflective_yellow
}
```

```
// SPHERE
object {
    sphere < 0.5, 0, 0>, 0.2
    reflective_yellow
}

// SPHERE
object {
    sphere < -0.5, 0, 0>, 0.2
    reflective_yellow
}

// CYLINDER
object {
    cylinder <0.0, -0.5, 0.0>, <0.0, 0.5, 0.0>, 0.2
    reflective_yellow
}

// SPHERE
object {
    sphere < 0, 0.5, 0>, 0.2
    reflective_yellow
}

// SPHERE
object {
    sphere < 0, -0.5, 0>, 0.2
    reflective_yellow
}
```

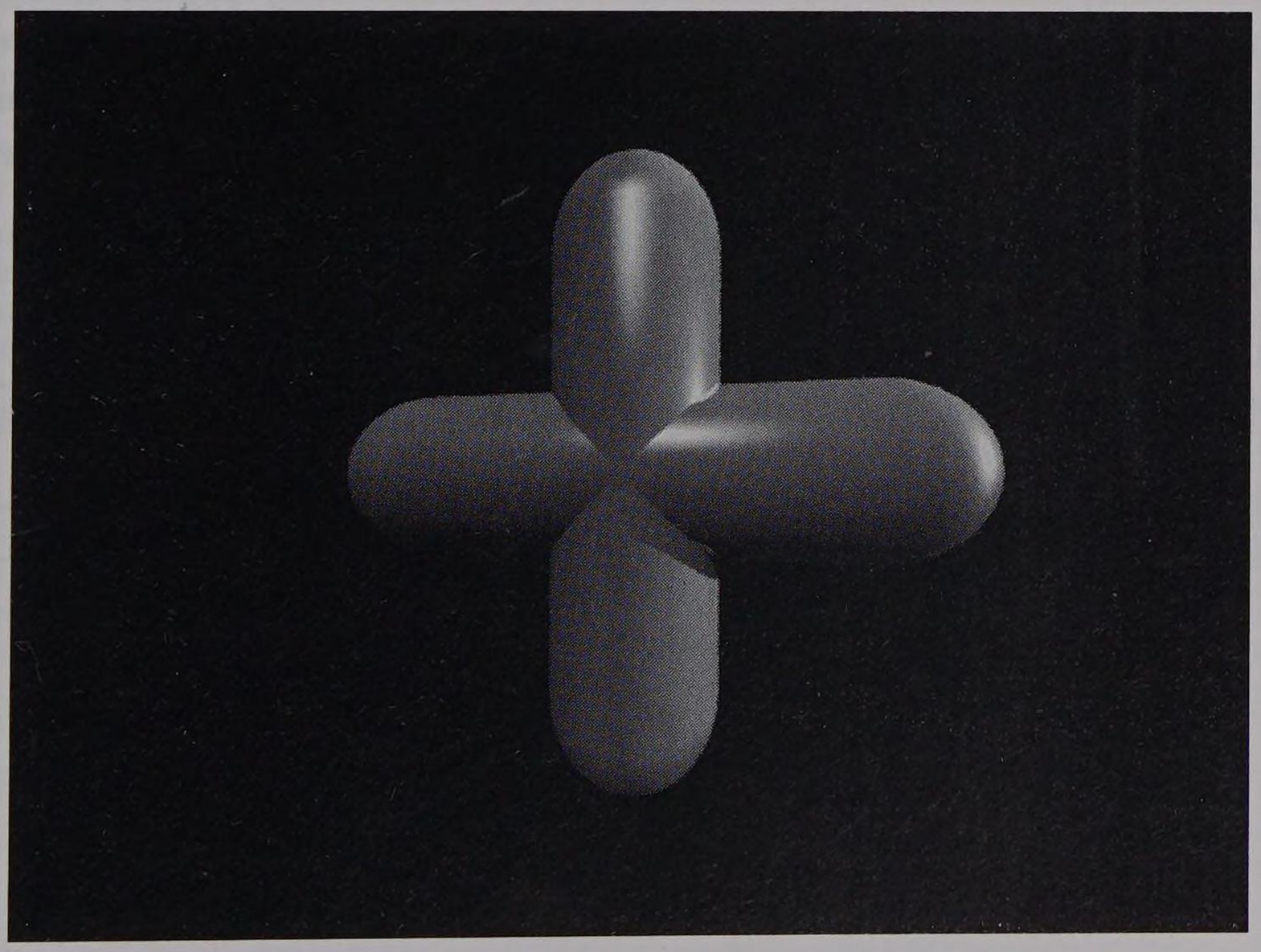


Figure 3.6

Using cylinders and spheres (CYLINDER.PI).

To cap a cylinder, use the following code:

```
define Cylinder_Y object {
                object { cylinder <0, -1, 0>, <0, 1, 0>, 1}
                    + object { disc <0, -1, 0>, <0, 1, 0>, 1 }
                    + object { disc <0,   1, 0>, <0, 1, 0>, 1 }
            }
```

Recall that the + sign refers to the Boolean operation `union`. Chapter 5, "Constructive Solid Geometry," explains Boolean operations in more detail.

Ellipsoid

Polyray doesn't have an ellipsoid object. You can, however, create one by scaling a sphere as in the following:

```
object {
     sphere < 0, 0, 0,>, 2
     scale <5, 2, 2>
     shiny_red
}
```

This creates an ellipsoid with the major axis along x.

Torus

A torus is a donut-shaped object with a major radius (the distance between the center of the torus and the center of the donut ring) and a minor radius (the radius of the ring). A torus also requires a normal vector, in the same fashion as disc objects (see fig 3.7). The syntax for a torus is

```
torus  rmaj, rmin, <xc, yc, zc>, <nx, ny, nz>
```

in which `xc`, `yc`, and `zc` are the coordinates of the center of the torus; `nx`, `ny`, `nz` is the normal vector; `rmaj` is the major radius; and `rmin` is the minor radius.

Listing 3.13 shows multiple torus objects rotated in the same fashion as in Listing 3.8 (see fig. 3.8 for the output).

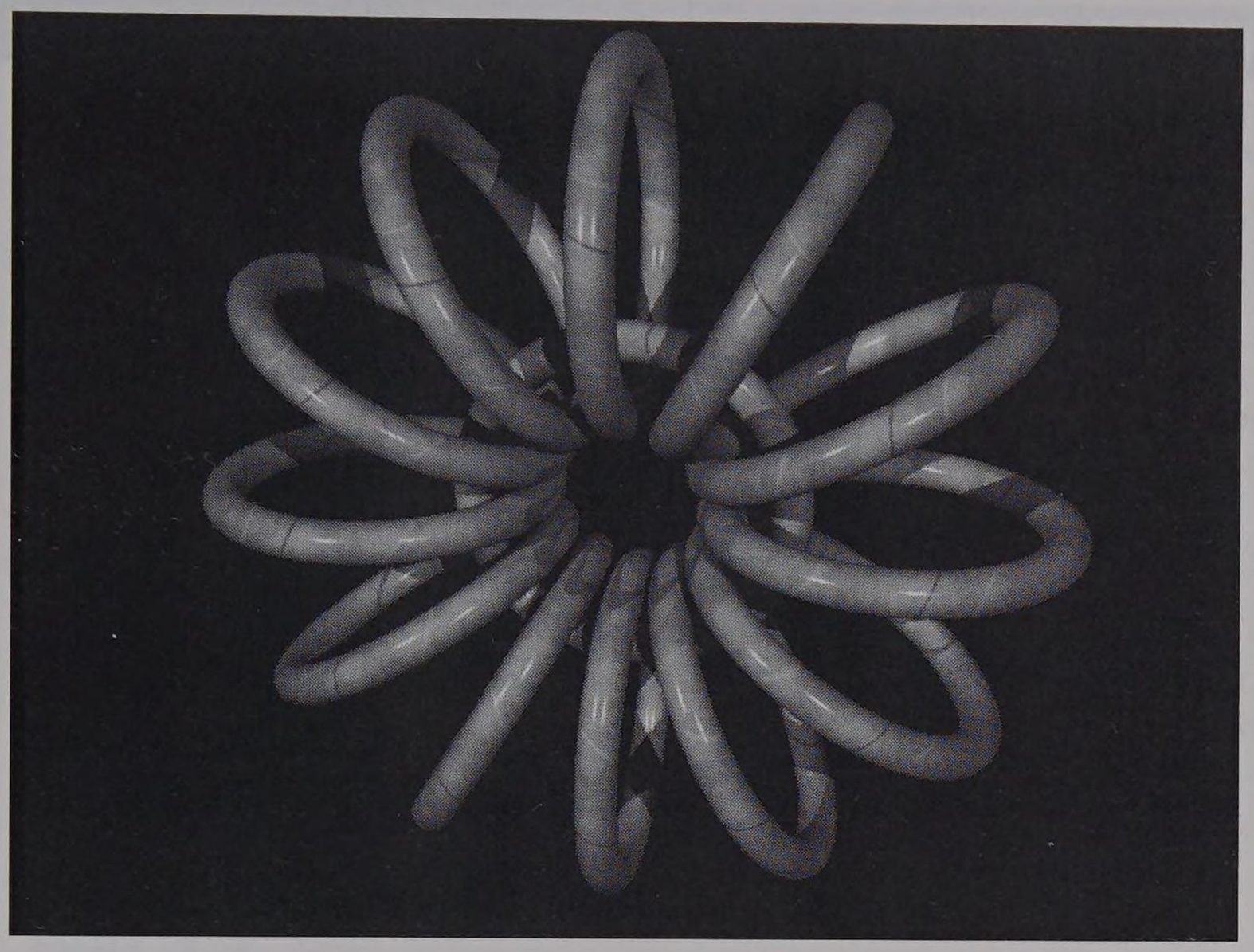


Figure 3.7

Torus Definition.

Listing 3.13. Torus example (TORUS.PI).

```
//TORUS.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

viewpoint {
 from      <1.0, 2.0, -5.0>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 angle 45
 hither 1
 resolution      100,100
 aspect          1.0
}

// LIGHT_SOURCE
    light White, <3, 3, -6>
// LIGHT_SOURCE
    light <-3, 3, -6>

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <90, 0, 0>
    Stone5
}
```

continues

Listing 3.13. Continued.

```
// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    translate <1.2437, 0.004338, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 30>
    translate <1.07491, 0.625606, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 60>
    translate <0.618092, 1.07924, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 90>
    translate <-0.004338, 1.2437, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 120>
    translate <-0.625606, 1.07491, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 150>
    translate <-1.07924, 0.618092, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 180>
    translate <-1.2437, -0.004338, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
```

```
    rotate <0, 0, 210>
    translate <-1.07491, -0.625606, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 240>
    translate <-0.618092, -1.07924, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 270>
    translate <0.004338, -1.2437, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 300>
    translate <0.625606, -1.07491, 0>
    Stone5
}

// TORUS
object {
    torus  0.8, 0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <0, 0, 330>
    translate <1.07924, -0.618092, 0>
    Stone5
}
```

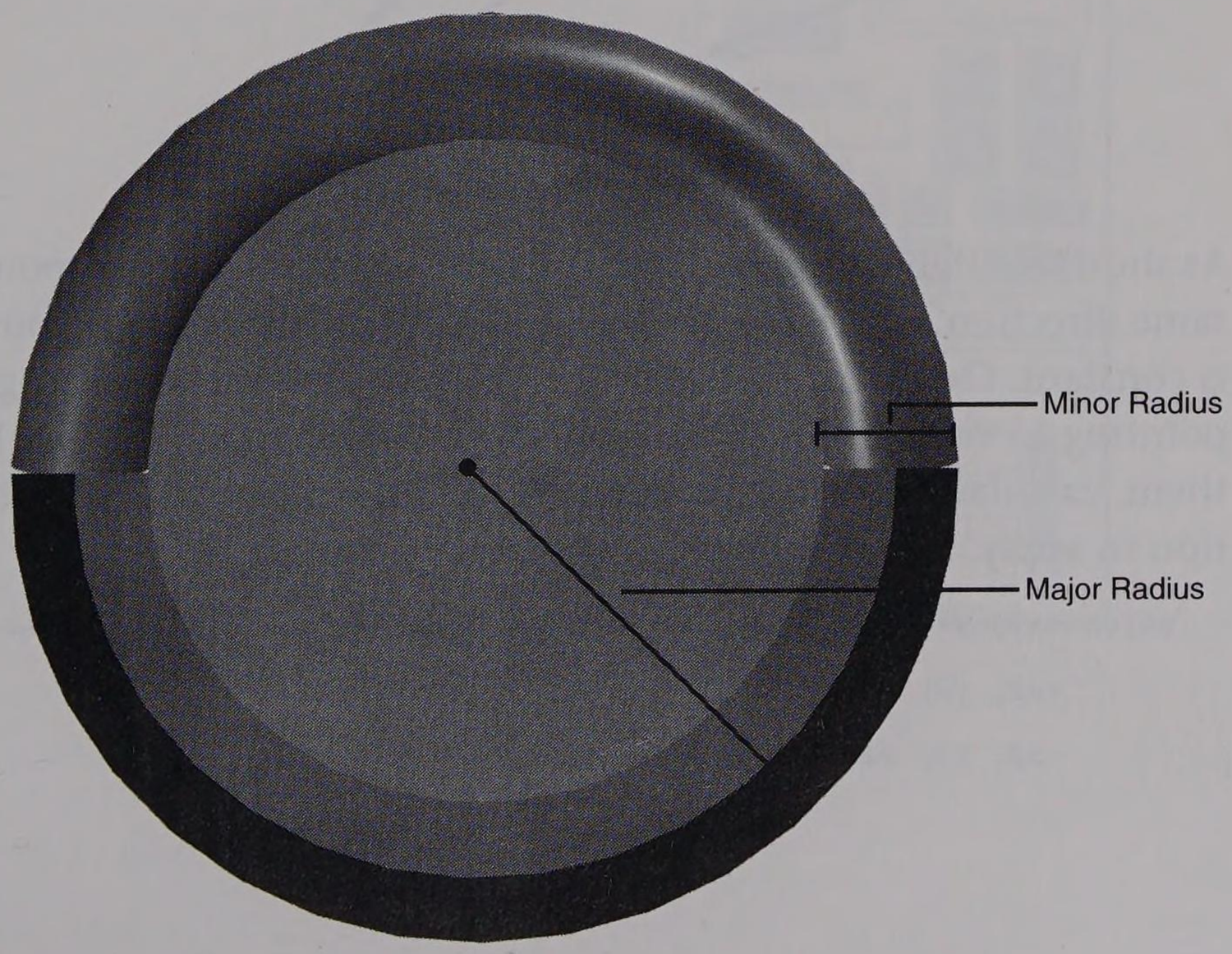


Figure 3.8

Torus example: rings on a ring (TORUS.PI).

Modeling Surfaces

Not all everyday objects can be modeled with those primitives presented in the previous chapter. In some cases, surfaces composed of triangular patches or polygons must be used to replicate a complicated object. In the following sections, you study techniques used to create and model surfaces.

Smooth Triangles

Smooth triangles or triangular patches are defined by three vertices and three normal vectors (one for each vertex). The normal vector is used to tell Polyray the orientation of the curvature at that vertex. This information is then used to compute the shades that will make the object look smooth instead of looking as if it was composed of flat triangles. Figure 3.9 shows an example of a flat triangle and a smooth one.

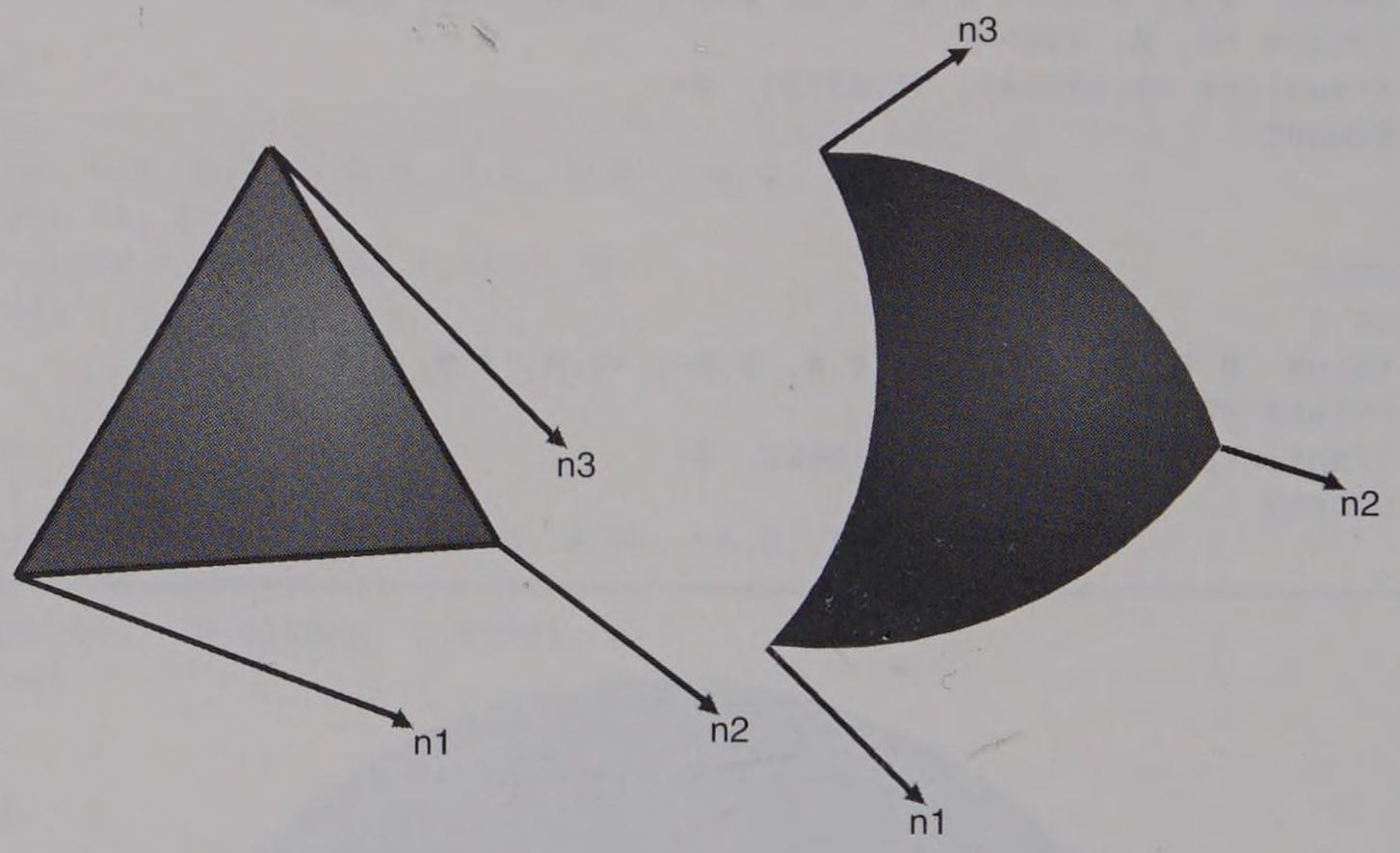


Figure 3.9
Flat and smooth triangles.

As shown in figure 3.9, flat triangles have all three normals pointing in the same direction. Therefore, the intensity of the color throughout the triangle is constant. On the other hand, the normals of a smooth triangle are pointing in various directions, so Polyray is able to interpolate between them, calculate what the curvature is, and determine what shading variation to apply. Polyray's syntax for smooth triangles is

```
patch <x1, y1, z1>, <n1x, n1y, n1z>,
      <x2, y2, z2>, <n2x, n2y, n2z>,
      <x3, y3, z3>, <n3x, n3y, n3z>
```

in which x1, y1, and z1 are the coordinates of vertex #1, and <n1x, n1y, n1z> is the normal vector of vertex #1, x2, y2, and z2 are the coordinates of vertex #2, and <n2x, n2y, n2z> is the normal vector of vertex #2 and finally x3, y3, and z3 are the coordinates of vertex #3, and <n3x, n3y, n3z> is the normal vector of vertex #3.

In the next section, you explore how smooth triangles are created.

Generating Smooth Triangles

Generating smooth triangle data by hand is extremely difficult, to say the least. It is not uncommon to see objects composed of thousands of triangles! Imagine typing all those values and their normals. There are utilities available that will help you create such objects. POVCAD (the 3-D modeler included with this book) can create them easily with just a few commands. Figures 3.10 and 3.11 show the two steps necessary to generate an object similar to a flower vase.

The first step is to create an outline of the object (see fig. 3.10). This outline consists of points interconnected with straight lines. You can do this by clicking the mouse in the places where the points are to be created.

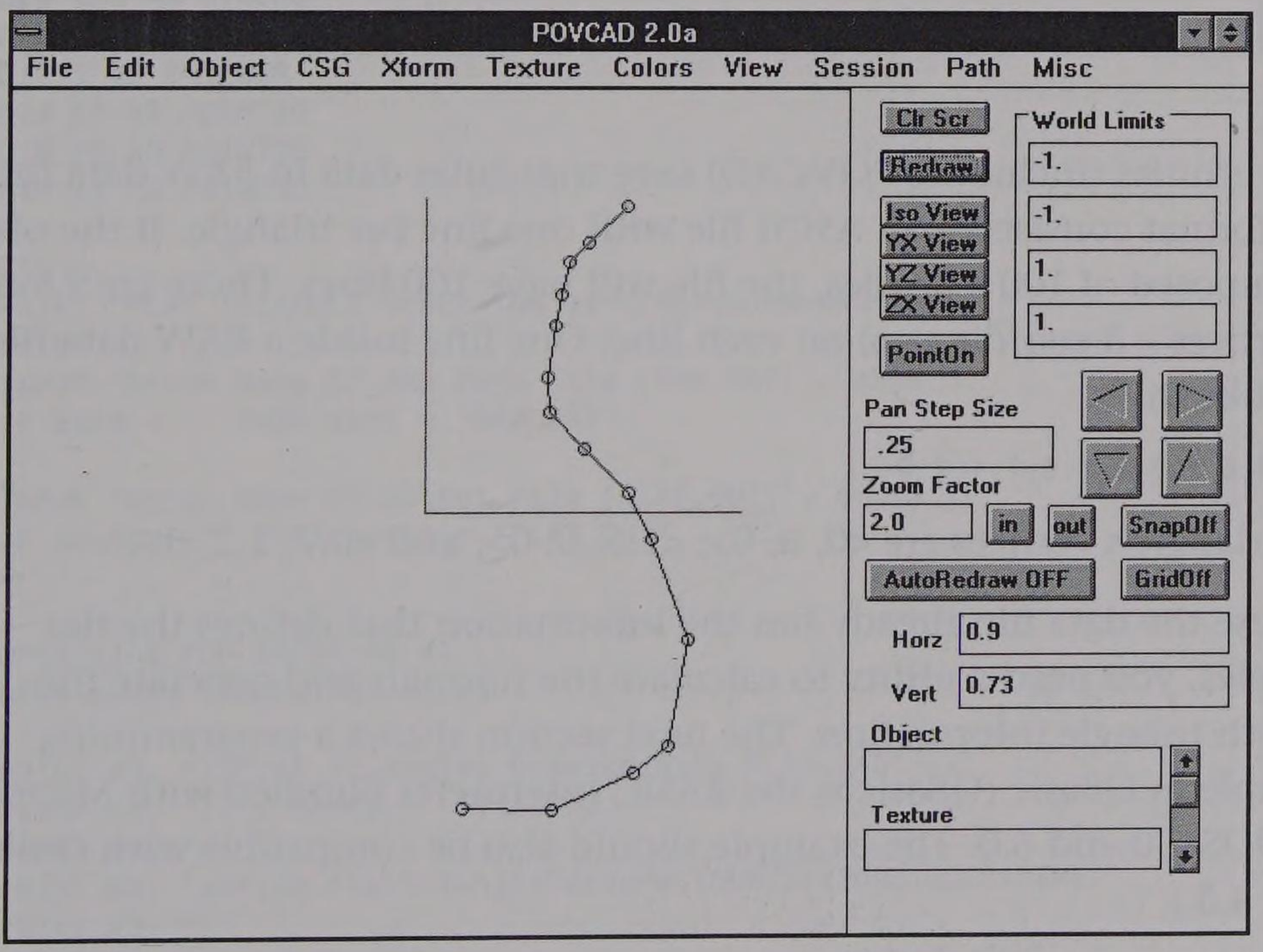


Figure 3.10

A vase outline created in POVCAD.

The next step is to select Sweep from the Object menu. The Sweep command in POVCAD is similar to the lathe object in Polyray. The contour is then rotated the necessary number of degrees. The result is an object that looks like a flower vase composed of flat triangles (see fig. 3.11).

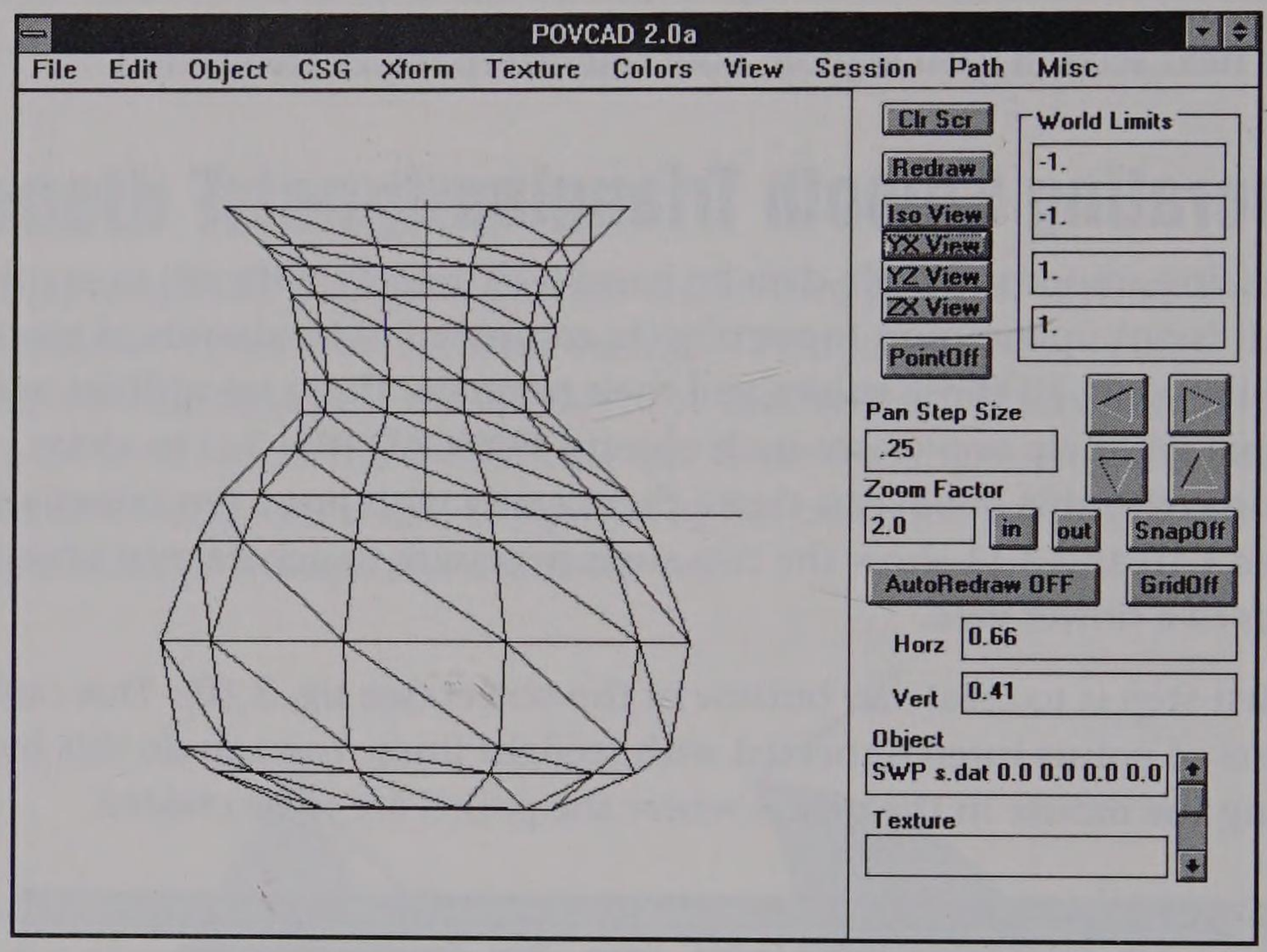


Figure 3.11

A complete vase created with the Sweep command.

Most utilities (including POVCAD) save triangular data in RAW data format. This format consists of an ASCII file with one line per triangle. If the object is composed of 100 triangles, the file will have 100 lines. There are 9 values (3 vertices × 3 coordinates) on each line. One line inside a RAW data file can look like this

```
0 0 0 1.5 0 0 0.7 1.2 0
```

The triangle's vertices are <0, 0, 0>; <1.5, 0, 0>; and <0.7, 1.2, 0>.

Because the data file already has the information that defines the flat triangles, you need a utility to calculate the normals and generate the smooth triangle information. The next section shows a programming example in QBasic (QBasic is the BASIC interpreter bundled with Microsoft's MS-DOS 5.0 and 6.0. The example should also be compatible with Quick Basic 4.5.)

Programming Example

In the previous section, you read that a computer program can be very helpful in creating smooth triangle information. This section presents two examples written in QBasic. The first program, FLAT.BAS, reads a RAW data file and creates an object composed of polygons. The data file that we are using is called RAW.DAT. It was created with POVCAD's Sweep command.

Listing 3.14. RAW data to polygon converter (FLAT.BAS).

```
' PROGRAM: FLAT.BAS
' by Alfonso Hermida 6/23/93
'
' This program reads a RAW data file <triangle data, 1 triangle per
' line> and outputs data as flat triangles for the Polyray ray
' tracer.
'
' This program is compatible with QBasic and Quick Basic 4.5.


TYPE POINT3D
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
END TYPE

DIM ain AS STRING
DIM aout AS STRING
DIM i AS INTEGER
DIM p1 AS POINT3D
DIM p2 AS POINT3D
DIM p3 AS POINT3D

CLS
PRINT "FLAT TRIANGLE GENERATOR by Alfonso Hermida"
PRINT
INPUT "Enter Name Of RAW Data File [RAW.DAT]", ain$
IF ain$ = "" THEN ain$ = "RAW.DAT"

INPUT "Enter Name Of Output File [FLAT.OUT]", aout$
IF aout$ = "" THEN aout$ = "FLAT.OUT"


OPEN ain$ FOR INPUT AS #1
OPEN aout$ FOR OUTPUT AS #3

PRINT #3, "//Flat Triangles Created With FLAT.BAS"
PRINT #3, "//"; DATE$; " "; TIME$
PRINT #3, ""
PRINT #3, "define FlatTriangleTexture YOUR/TEXTURE/GOES/HERE!"
PRINT #3, " "
PRINT #3, ""
PRINT #3, "object {"
```

continues

Listing 3.14. Continued.

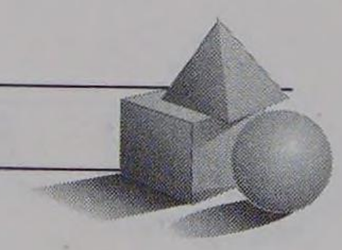
```
i = 0
DO WHILE NOT EOF(1)

  INPUT #1, p1.x, p1.y, p1.z
  INPUT #1, p2.x, p2.y, p2.z
  INPUT #1, p3.x, p3.y, p3.z

  i = i + 1
  IF i > 1 THEN PRINT #3, "+";
  PRINT #3, "   object {"
  PRINT #3, "      polygon 3, "
  PRINT #3, "                <";
  PRINT #3, USING "#####.###"; p1.x;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p1.y;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p1.z;
  PRINT #3, ">,"
  PRINT #3, "                <";
  PRINT #3, USING "#####.###"; p2.x;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p2.y;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p2.z;
  PRINT #3, ">,"
  PRINT #3, "                <";
  PRINT #3, USING "#####.###"; p3.x;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p3.y;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p3.z;
  PRINT #3, ">"
  PRINT #3, "   }"

LOOP
PRINT #3, "    FlatTriangleTexture"
PRINT #3, "}"
CLOSE #3
CLOSE #1
PRINT
PRINT "Done!"
```

The program opens a RAW data file (in this case, the default name is RAW.DAT) and saves each vertex in the variables `p1`, `p2`, and `p3`. The three points are then written to a file to comply with Polyray's polygon format. At the top of the file is the following statement:

```
define FlatTriangleTexture YOUR/TEXTURE/GOES/HERE!
```

Replace the expression `YOUR/TEXTURE/GOES/HERE!` with the corresponding texture (finish) or color. For example:

```
define FlatTriangleTexture marble_finish
```



In the preceding line, the expression was replaced with `marble_finish`. This texture is used for all the polygons generated in that file. The token name `FlatTriangleTexture` appears at the end of the output file.

Generating flat triangles is easy if the RAW data file is available. The next program, named SMOOTH.BAS (see Listing 3.15), reads the same RAW data file and creates smooth triangles or patches. To calculate the normals of the smooth object, the program does the following:

1. Calculate the normal of each vertex and save that information. Because the triangles are initially flat, all the normals of a given triangle must be equal.
2. Check which vertices are shared between triangles. If a vertex is shared by other triangles, add all the normals and calculate an average normal for the given vertex.
3. Repeat step 2 until all vertices are checked.
4. Output the data.

The normal of a triangle (or plane) is calculated in the subroutine named `NormalVector`. I will not get into the details of how to calculate a normal; however, you might want to review this process in a calculus book. The subroutine is very short and easily understood.

Listing 3.15. Smooth triangle generator (SMOOTH.BAS).

```
' PROGRAM: SMOOTH.BAS
' by Alfonso Hermida 6/23/93
'
' This program reads a RAW data file <triangle data, 1 triangle
' per line> and outputs data as smooth triangles for the Polyray
' ray tracer.
'
' NOTE: The array sizes of
'
' DIM Vertex(3300) AS POINT3D
' DIM Normal(3300) AS POINT3D
' must be big enough to hold all your input data.
' If you can't run the program, experiment by decreasing the size.

' This program is compatible with QBasic and Quick Basic 4.5.

DECLARE SUB NormalVector (pc AS ANY, p1 AS ANY, p2 AS ANY, pn AS ANY)

TYPE POINT3D
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
```

continues

Listing 3.15. Continued.

```
END TYPE

 DIM a AS STRING
 DIM TOL AS SINGLE
 DIM i, j, k, RST, total, count AS INTEGER
 DIM p1 AS POINT3D
 DIM p2 AS POINT3D
 DIM p3 AS POINT3D
 DIM pnt AS POINT3D
 DIM pn AS POINT3D
 DIM pnormal AS POINT3D
 DIM Vertex(3300) AS POINT3D  'Change this number if necessary
 DIM Normal(3300) AS POINT3D  'Change this number if necessary
 DIM Magnitude AS SINGLE

 TOL = .0005#
 CLS
 PRINT "SMOOTH TRIANGLE GENERATOR by Alfonso Hermida"
 PRINT
 INPUT "Enter Name Of RAW Data File [RAW.DAT]", a$
 IF a$ = "" THEN a$ = "RAW.DAT"

 OPEN a$ FOR INPUT AS #1
 count = 1
 DO WHILE NOT EOF(1)

    INPUT #1, Vertex(count).x
    INPUT #1, Vertex(count).y
    INPUT #1, Vertex(count).z
    INPUT #1, Vertex(count + 1).x
    INPUT #1, Vertex(count + 1).y
    INPUT #1, Vertex(count + 1).z
    INPUT #1, Vertex(count + 2).x
    INPUT #1, Vertex(count + 2).y
    INPUT #1, Vertex(count + 2).z
    p1 = Vertex(count)
    p2 = Vertex(count + 1)
    p3 = Vertex(count + 2)

    'Calculate the triangle normal
    CALL NormalVector(p1, p2, p3, pnormal)

    'Copy the normal to each vertex
    Normal(count) = pnormal
    Normal(count + 1) = pnormal
    Normal(count + 2) = pnormal

    'Increment vertex counter
    count = count + 3

 LOOP
 count = count - 1
 CLOSE #1


 PRINT "Results:"
```

```
PRINT "Points    = "; count
PRINT "Triangles = "; count / 3
PRINT

INPUT "Enter Name Of Output File [SMOOTH.OUT]", a$
IF a$ = "" THEN a$ = "SMOOTH.OUT"
PRINT "Please wait..."

OPEN a$ FOR OUTPUT AS #3
PRINT #3, "//Smooth Triangles Created With SMOOTH.BAS"
PRINT #3, "//"; DATE$; " "; TIME$
PRINT #3, ""
PRINT #3, "define SmoothTriangleTexture YOUR/TEXTURE/GOES/HERE!"
PRINT #3, " "
PRINT #3, ""
PRINT #3, "object {"
RST = 0
FOR i = 1 TO count
 LOCATE 10, 1
 PRINT i
  p1 = Vertex(i)
  total = 0
  RST = RST + 1
  IF RST = 1 THEN
      IF i > 1 THEN PRINT #3, "+";
      PRINT #3, "   object { patch"
  END IF
  pn.x = 0
  pn.y = 0
pn.z = 0
total = 0
FOR k = 1 TO count
   ' Check And Add All Normals Of A Given Vertex
  IF (ABS(p1.x - Vertex(k).x) < TOL AND
   ➥ABS(p1.y - Vertex(k).y) < TOL AND
   ➥ABS(p1.z - Vertex(k).z) < TOL) THEN
  pn.x = pn.x + Normal(k).x
  pn.y = pn.y + Normal(k).y
  pn.z = pn.z + Normal(k).z
  total = total + 1
  END IF
NEXT

' Compute Normal Average
pn.x = pn.x / total
pn.y = pn.y / total
pn.z = pn.z / total

' Find The Magnitude Of The Normal
Magnitude = (pn.x ^ 2 + pn.y ^ 2 + pn.z ^ 2) ^ (.5)

' Make the Normal A Unit Vector
pn.x = pn.x / Magnitude
pn.y = pn.y / Magnitude
pn.z = pn.z / Magnitude

PRINT #3, "                <";
PRINT #3, USING "#####.###"; p1.x;
```

continues

Listing 3.15. Continued.

```
    PRINT #3, ",";
    PRINT #3, USING "#####.###"; p1.y;
    PRINT #3, ",";
    PRINT #3, USING "#####.###"; p1.z;
    PRINT #3, ">, < ";
    PRINT #3, USING "#####.###"; pn.x;
    PRINT #3, ", ";
    PRINT #3, USING "#####.###"; pn.y;
    PRINT #3, ", ";
    PRINT #3, USING "#####.###"; pn.z;
    PRINT #3, " >";
    IF RST < 3 THEN PRINT #3, ","
    IF RST = 3 THEN
        PRINT #3, ""
        PRINT #3, "   }"
        RST = 0
    END IF
  NEXT
  PRINT #3, "    SmoothTriangleTexture"
  PRINT #3, "}"
  CLOSE #3
  PRINT ""
  PRINT "Done!"

SUB NormalVector (pc AS POINT3D, p1 AS POINT3D,
   ➥p2 AS POINT3D, pn AS POINT3D)
DIM V1 AS POINT3D
DIM V2 AS POINT3D

  V1.x = (p1.x - pc.x)
  V1.y = (p1.y - pc.y)
  V1.z = (p1.z - pc.z)

  V2.x = (p2.x - pc.x)
  V2.y = (p2.y - pc.y)
  V2.z = (p2.z - pc.z)

' calc normal vector N = Ai+Bj+Ck
  pn.x = V1.y * V2.z - V1.z * V2.y
  pn.y = -(V1.x * V2.z - V1.z * V2.x)
  pn.z = V1.x * V2.y - V1.y * V2.x

END SUB
```

Figures 3.12 and 3.13 show the ray-traced images of the output generated by FLAT.BAS and SMOOTH.BAS, respectively.

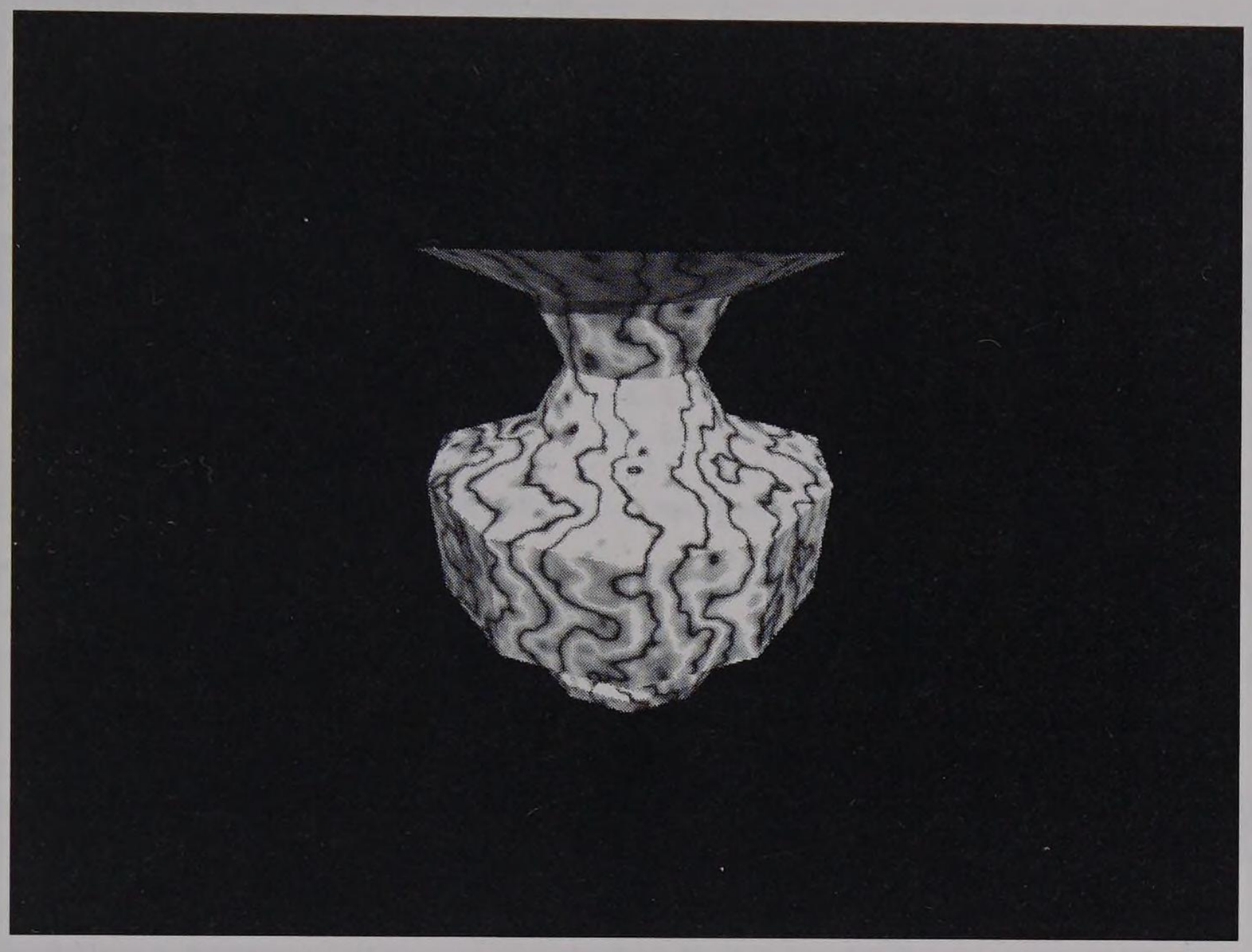


Figure 3.12

The output from data generated by FLAT.BAS.

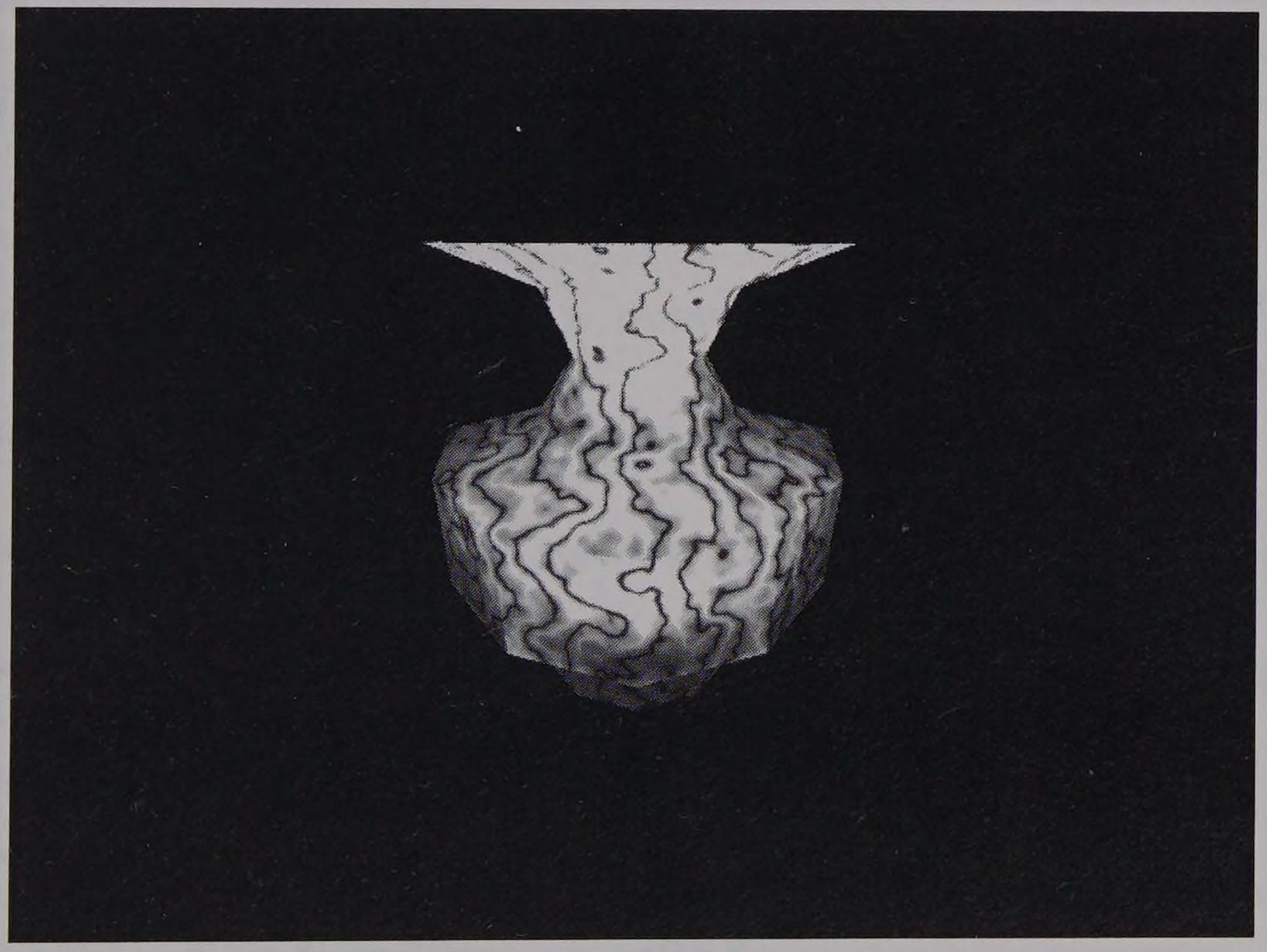


Figure 3.13

The output from data generated by SMOOTH.BAS.

Polygons

Polygons are collections of points that are interconnected to create a flat surface. Recall how in previous examples we used polygons to create a floor. The syntax is

```
polygon n,
     < vertex1.x, vertex1.y, vertex1.z>,
     < vertex2.x, vertex2.y, vertex2.z>,
     < vertex3.x, vertex3.y, vertex3.z>,
          .... etc. ...
     < vertexn.x, vertexn.y, vertexn.z>
```

where `n` is the number of vertices.

Check previous examples, for example PLANES.PI, for the use of polygons.

Lathe

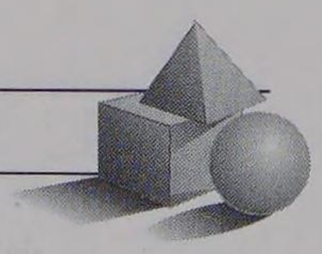
Lathe objects are a collection of points that define the contour of a symmetric object. The contour is then revolved about an axis creating a *solid of revolution*. There are two types of lathe objects. Type 1 connects the points with line segments, whereas type 2 uses a smooth curve (spline). The contour can be rotated about any axis by defining a direction vector. To rotate about the y axis, use <0, 1, 0>. To rotate about the x or z axis, use <1, 0, 0> or <0, 0, 1>, respectively.

The syntax is

```
object {
    lathe  type , axis,  n ,
           <p1.x, p1.y, p1.z>,
           <p2.x, p2.y, p2.z>,
..............etc...........
           <pn.x, pn.y, pn.z>
}
```

in which

`type`	Either 1 (faceted finish) or 2 (smooth finish).
`axis`	Axis of rotation, i.e. <0, 1, 0> is about the y axis.
`n`	Number of points that define the curve.
`<pi.x, pi.y, pi.z>`	The `i`th point's coordinates.



The program LATHE.BAS is included on the disk and works in the same way as FLAT.BAS (see Listing 3.16). It opens a data file and creates another file that follows Polyray's format, in this case for lathe objects.

Listing 3.16. Program LATHE.BAS.

```
' PROGRAM: LATHE.BAS
' by Alfonso Hermida 6/23/93
'
' This program reads a list of 3D points that defines a contour
' and outputs data as a LATHE object for the Polyray ray tracer.
'
' This program is compatible with QBasic and Quick Basic 4.5.


TYPE POINT3D
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
END TYPE

 DIM ain AS STRING
 DIM aout AS STRING
 DIM LatheType AS INTEGER
 DIM Total AS INTEGER
 DIM Direction AS STRING
 DIM i AS INTEGER
DIM p1 AS POINT3D

CLS
PRINT "LATHE OBJECT GENERATOR by Alfonso Hermida"
PRINT
INPUT "Enter Name Of RAW Data File [POINTS.DAT]", ain$
IF ain$ = "" THEN ain$ = "POINTS.DAT"

INPUT "Enter Name Of Output File [LATHE.OUT]", aout$
IF aout$ = "" THEN aout$ = "LATHE.OUT"

INPUT "Select A Type: (1) Line Segments (2) Smooth (Spline)  [2]"; LatheType
IF LatheType = 0 THEN LatheType = 2

INPUT "Enter Direction Of Axis Of Rotation [<0, 1, 0>]"; Direction$
IF Direction$ = "" THEN Direction$ = "<0, 1, 0>"

'Check Total Number Of Points
i = 0
OPEN ain$ FOR INPUT AS #1
DO WHILE NOT EOF(1)
  INPUT #1, p1.x, p1.y, p1.z
  i = i + 1
LOOP
CLOSE #1
Total = i

OPEN ain$ FOR INPUT AS #1
```

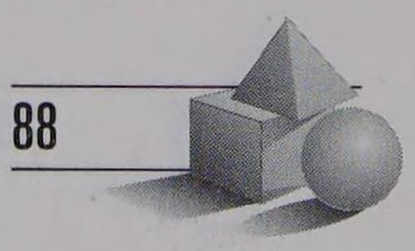
continues

Listing 3.16. Continued.

```
OPEN aout$ FOR OUTPUT AS #3

PRINT #3, "//Lathe Object Created With LATHE.BAS"
PRINT #3, "//"; DATE$; " "; TIME$
PRINT #3, ""
PRINT #3, "define LatheTexture YOUR/TEXTURE/GOES/HERE!"
PRINT #3, ""
PRINT #3, ""
PRINT #3, "object {"
PRINT #3, "    lathe "; LatheType; ", "; Direction$; ", "; Total; ","

i = 0
DO WHILE NOT EOF(1)

  INPUT #1, p1.x, p1.y, p1.z

  i = i + 1
  PRINT #3, "          <";
  PRINT #3, USING "#####.###"; p1.x;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p1.y;
  PRINT #3, ",";
  PRINT #3, USING "#####.###"; p1.z;
  PRINT #3, ">";
  IF i < Total THEN
     PRINT #3, ","
  ELSE
     PRINT #3, ""
   END IF
 LOOP
 PRINT #3, "    LatheTexture"
 PRINT #3, "}"
 CLOSE #3
 CLOSE #1
 PRINT
 PRINT "Done!"
```

The data file used was PAWN.DAT, which has the following format:

```
PAWN.DAT

0              5.44     0
0.672269       5.15401  0
0.87395        4.52928  0
0.638655       3.97397  0
0.336134       3.86985  0
0.806723       3.80043  0
1.10924        3.59219  0
1.14286        3.27983  0
0.705882       3.10629  0
0.336134       3.10629  0
0.470588       2.44685  0
0.638655       1.96095  0
0.941176       1.54447  0
1.31092        1.09328  0
```



```
2.01681          0.67679     0
2.38655          0.295011    0
2.38655         -0.225597    0
2.01681         -0.537961    0
2.52101         -0.607375    0
2.92437         -0.885033    0
2.85714         -1.30152     0
2.31933         -1.44035     0
1.64706         -1.54447     0
0               -1.6         0
```

The format is simple: there is one 3-D point per line to define the contour of a chess piece. The program LATHE.BAS read this data and creates an output similar to Listing 3.17.

Listing 3.17. The output from LATHE.BAS (LATHE.PI).

```
// LATHE.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <0.0, 4.0, -9.0>
 at        <0.0, 1.0, 0.0>
 up        <0.0, 1.0, 0.0>
 angle 80
 resolution      100,100
 aspect          1.00
}

background White

// LIGHT_SOURCE
    light White, <0, 1, -6>
// LIGHT_SOURCE
    light <-3, 3, -6>
    light < 3, 3, -6>


//Lathe Object Created With LATHE.BAS
//06-24-1993 10:38:17

define LatheTexture shiny_red

define FlatPawn
object {
    lathe  1 , <0, 1, 0>,  24 ,
           <    0.000,    5.440,    0.000>,
           <    0.672,    5.154,    0.000>,
           <    0.874,    4.529,    0.000>,
           <    0.639,    3.974,    0.000>,
           <    0.336,    3.870,    0.000>,
           <    0.807,    3.800,    0.000>,
           <    1.109,    3.592,    0.000>,
```

continues

Listing 3.17. Continued.

```
            <    1.143,    3.280,    0.000>,
            <    0.706,    3.106,    0.000>,
            <    0.336,    3.106,    0.000>,
            <    0.471,    2.447,    0.000>,
            <    0.639,    1.961,    0.000>,
            <    0.941,    1.544,    0.000>,
            <    1.311,    1.093,    0.000>,
            <    2.017,    0.677,    0.000>,
            <    2.387,    0.295,    0.000>,
            <    2.387,   -0.226,    0.000>,
            <    2.017,   -0.538,    0.000>,
            <    2.521,   -0.607,    0.000>,
            <    2.924,   -0.885,    0.000>,
            <    2.857,   -1.302,    0.000>,
            <    2.319,   -1.440,    0.000>,
            <    1.647,   -1.544,    0.000>,
            <    0.000,   -1.600,    0.000>
        LatheTexture
}

define SmoothPawn
object {
    lathe  2 , <0, 1, 0>,  24 ,
            <    0.000,    5.440,    0.000>,
            <    0.672,    5.154,    0.000>,
            <    0.874,    4.529,    0.000>,
            <    0.639,    3.974,    0.000>,
            <    0.336,    3.870,    0.000>,
            <    0.807,    3.800,    0.000>,
            <    1.109,    3.592,    0.000>,
            <    1.143,    3.280,    0.000>,
            <    0.706,    3.106,    0.000>,
            <    0.336,    3.106,    0.000>,
            <    0.471,    2.447,    0.000>,
            <    0.639,    1.961,    0.000>,
            <    0.941,    1.544,    0.000>,
            <    1.311,    1.093,    0.000>,
            <    2.017,    0.677,    0.000>,
            <    2.387,    0.295,    0.000>,
            <    2.387,   -0.226,    0.000>,
            <    2.017,   -0.538,    0.000>,
            <    2.521,   -0.607,    0.000>,
            <    2.924,   -0.885,    0.000>,
            <    2.857,   -1.302,    0.000>,
            <    2.319,   -1.440,    0.000>,
            <    1.647,   -1.544,    0.000>,
            <    0.000,   -1.600,    0.000>
        LatheTexture
}

object {
  FlatPawn
  translate <-4, 0, 0>
}

object {
 SmoothPawn
 translate <4, 0, 0>
}
```

In LATHE.PI, two lathe objects were created with the same data points. Each uses a different type of lathe finish. The first one uses type 1, whereas the second uses type 2. Figure 3.14 shows both objects.

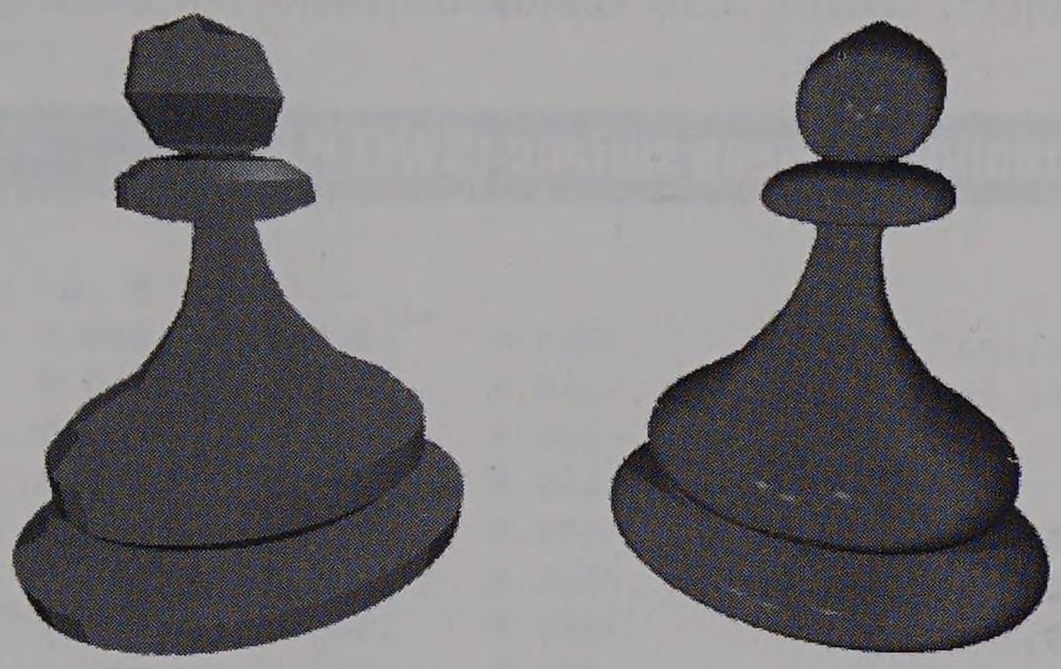


Figure 3.14
Chess pawns created with the lathe command (LATHE.PI).

Sweep/Extruded

Sweep surfaces are similar in syntax to lathe surfaces with some minor differences. You must replace the word `lathe` with the word `sweep`. Also, the direction vector defines a translation rather than a rotation axis.

The syntax is

```
object {
    sweep  type , axis,  n ,
           <p1.x, p1.y, p1.z>,
           <p2.x, p2.y, p2.z>,
.............etc..........
           <pn.x, pn.y, pn.z>
}
```

in which

`type`	Either 1 (faceted finish) or 2 (smooth finish).
`axis`	Axis of translation, i.e. <0, 1, 0> is along the Y axis.
`n`	Number of points that define the curve.
`<pi.x, pi.y, pi.z>`	The `i`th point's coordinates.

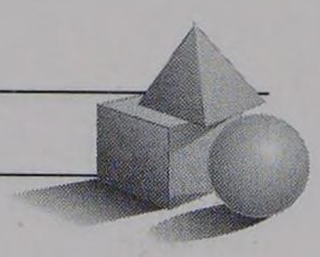
The magnitude of the translation vector is very important, because it determines the total displacement (extrusion) of the curve along the vector. To calculate the magnitude of a vector, use the following formula:

given <a, b, c>, $\quad$ magnitude $= \sqrt{a^2 + b^2 + c^2}$

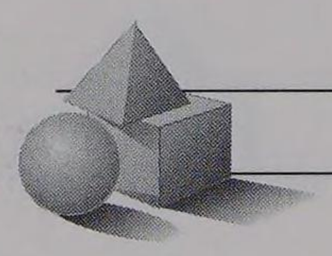
As a small experiment, we use the same contour data from PAWN.DAT to create a sweep object. Listing 3.18 is a modification of Listing 3.16.

Listing 3.18. Creating a sweep surface (SWEEP.PI).

```
// SWEEP.PI

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <0.0, 3.0, -9.0>
 at        <1.0, 1.0, 0.0>
 up        <0.0, 1.0, 0.0>
 angle 80
 resolution      100,100
 aspect          1.00
}

background White

// LIGHT_SOURCE
    light White, <0, 1, -6>
// LIGHT_SOURCE
    light <-3, 3, -6>
    light < 3, 3, -6>


define SweepTexture shiny_red

define FlatPawn
object {
    sweep  1 , <0, 0, 3>,  24 ,
          <    0.000,    5.440,    0.000>,
          <    0.672,    5.154,    0.000>,
          <    0.874,    4.529,    0.000>,
          <    0.639,    3.974,    0.000>,
          <    0.336,    3.870,    0.000>,
          <    0.807,    3.800,    0.000>,
          <    1.109,    3.592,    0.000>,
          <    1.143,    3.280,    0.000>,
          <    0.706,    3.106,    0.000>,
          <    0.336,    3.106,    0.000>,
          <    0.471,    2.447,    0.000>,
          <    0.639,    1.961,    0.000>,
          <    0.941,    1.544,    0.000>,
```



```
            <    1.311,    1.093,    0.000>,
            <    2.017,    0.677,    0.000>,
            <    2.387,    0.295,    0.000>,
            <    2.387,   -0.226,    0.000>,
            <    2.017,   -0.538,    0.000>,
            <    2.521,   -0.607,    0.000>,
            <    2.924,   -0.885,    0.000>,
            <    2.857,   -1.302,    0.000>,
            <    2.319,   -1.440,    0.000>,
            <    1.647,   -1.544,    0.000>,
            <    0.000,   -1.600,    0.000>
    SweepTexture
}

define SmoothPawn
object {
    sweep  2 , <0, 0, 3>,  24 ,
            <    0.000,    5.440,    0.000>,
            <    0.672,    5.154,    0.000>,
            <    0.874,    4.529,    0.000>,
            <    0.639,    3.974,    0.000>,
            <    0.336,    3.870,    0.000>,
            <    0.807,    3.800,    0.000>,
            <    1.109,    3.592,    0.000>,
            <    1.143,    3.280,    0.000>,
            <    0.706,    3.106,    0.000>,
            <    0.336,    3.106,    0.000>,
            <    0.471,    2.447,    0.000>,
            <    0.639,    1.961,    0.000>,
            <    0.941,    1.544,    0.000>,
            <    1.311,    1.093,    0.000>,
            <    2.017,    0.677,    0.000>,
            <    2.387,    0.295,    0.000>,
            <    2.387,   -0.226,    0.000>,
            <    2.017,   -0.538,    0.000>,
            <    2.521,   -0.607,    0.000>,
            <    2.924,   -0.885,    0.000>,
            <    2.857,   -1.302,    0.000>,
            <    2.319,   -1.440,    0.000>,
            <    1.647,   -1.544,    0.000>,
            <    0.000,   -1.600,    0.000>
    SweepTexture
}

object {
  FlatPawn
  rotate <0, 55, 0>
  translate <-3, 0, 0>
}

object {
 SmoothPawn
 rotate <0, 70, 0>
 translate <3, 0, 0>
}
```



Again, two objects are created (see fig 3.15). One of the objects uses type 1 (faceted), and the other uses type 2 (smooth).

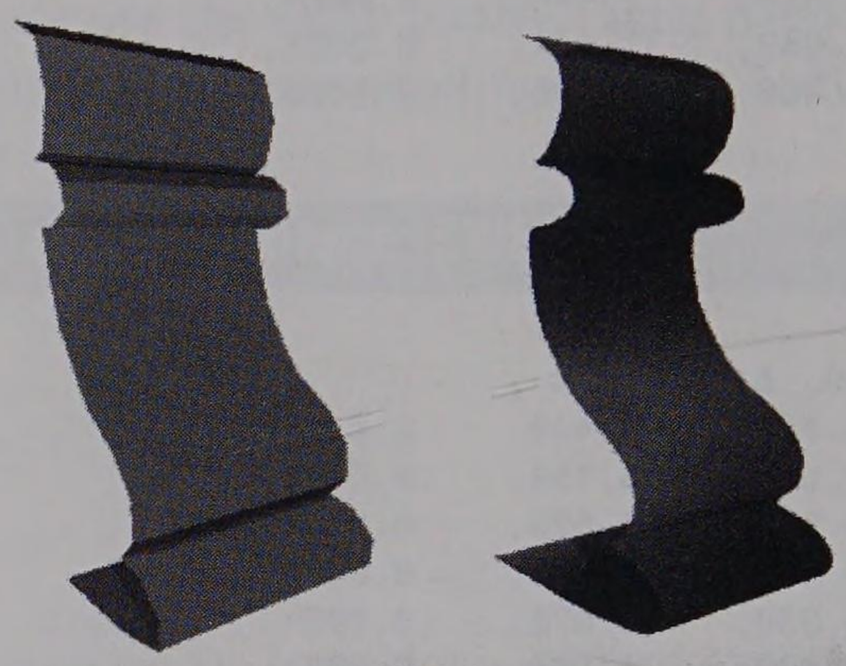


Figure 3.15
Extruded chess pawn curves (SWEEP.PI).

You can easily modify the program LATHE.BAS to create sweep objects. This programming exercise is left to the reader.

Bézier Patches

Bézier patches consist of 16 control points arranged in a 4 × 4 matrix or array. By displacing these control points, the patch can be molded. The syntax is

```
bezier s, f, u, v,
     <x0, y0, z0>,
     <x1, y1, z1>,
          .........,
    <x15, y15, z15>,
```

in which

s	is the subdivision control and refers to how the data is stored:
s = 1	uses less memory but takes longer to generate
s = 2	uses more memory but takes less time

`f`	Sets the flatness value. Lowering this number increases the amount of subdivisions as determined by `u` and `v`. The suggested value of `f` is 0.05.
`u,v`	Number of subdivisions to be performed on the local x and y axis of the patch. Increasing this number forces the surface to be smoother at the expense of memory and computation time. Suggested values: `u` = 3, `v` = 3.

Listing 3.19 shows an example of two surfaces created with Bézier patches. The `SheetBezier` object was created by deforming a flat Bézier patch, and the `CylinderBezier` object was created by deforming four Bézier patches initially arranged as a cylinder. The ends of the cylinder were stretched outward (see fig. 3.16).

Observe the interesting effect of applying a checkered finish to the `SheetBezier` surface. One of the tiles is made of stone and the other is just the color `matte_black`. Another interesting feature is that the `CylinderBezier` surface is four Bézier patches combined into one by using Boolean operations (+).

Listing 3.19. Bézier patches (BEZIER.PI).

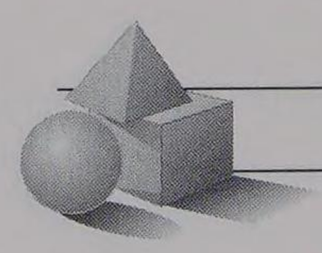
```
// BEZIER.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

viewpoint {
 from      <-5.0, 5.0, -30.0>
 at        <0.0, 0.0, 0.0>
 up        <0.0, 1.0, 0.0>
 angle 85
 resolution      100,100
 aspect          1.0
}

background White

// LIGHT_SOURCE
    light White, <0, 1, -6>
    light <0,-6,-6>
// LIGHT_SOURCE
    light <-20, 5, 0>
    light < 3, 3, -6>

define SheetBezier object {
                bezier  2, 0.05,  3,  3,
     <-7.441097,     0.175513,     8.736674>,
```

continues

Listing 3.19. Continued.

```
                <0.775496,    -1.090713,     4.650486>,
                <2.999994,    -0.058649,     4.433383>,
                <9.859163,    -2.451552,    -1.003611>,
               <-2.348895,    -0.067749,     2.807758>,
               <-0.332253,     0.867879,     2.610940>,
                <2.060343,     2.696604,     8.953340>,
                <4.012816,     2.883789,     2.186876>,
               <-3.204923,     1.445780,     1.231671>,
               <-0.367755,     1.198388,    -4.274309>,
                <1.036218,     3.413472,     0.817750>,
                <3.156788,     4.397318,     0.610789>,
               <-8.325093,     6.300124,     4.751141>,
               <-2.203973,     4.177238,    -0.835203>,
                <0.000567,     5.244590,    -1.089052>,
                <4.265499,     6.530803,    -2.477069>
                        }

define b21      object {
                  bezier  2, 0.05,  3,  3,
            <-13.157171,     3.747712,   -13.266072>,
            <-15.994769,    -2.360541,   -13.257848>,
             <-9.888506,    -9.988533,   -12.331910>,
             <-6.020190,   -10.754260,   -12.339270>,
            <-10.325383,     1.543393,    -0.767010>,
            <-10.806159,    -1.222832,    -0.967095>,
             <-8.392786,    -4.410553,    -1.158496>,
             <-5.681259,    -4.927205,    -1.098878>,
            <-10.599442,     1.220823,     1.557361>,
            <-10.882533,    -1.412619,     1.840276>,
             <-8.590908,    -4.663777,     1.682831>,
             <-5.818507,    -5.148710,     1.725471>,
            <-14.989909,     2.110008,     5.359019>,
            <-15.732432,    -1.233681,     3.958136>,
            <-16.821856,    -5.317152,     3.020572>,
             <-7.121402,    -8.133587,     3.127131>
                        }

define  b22     object {
                    bezier  2, 0.05,  3,  3,
            <-6.020190,   -10.754260,   -12.339270>,
            <-2.151873,   -11.519988,   -12.346631>,
             <0.620599,    -6.774581,   -12.403907>,
             <3.010568,    -2.939457,   -12.403823>,
            <-5.681259,    -4.927205,    -1.098878>,
            <-2.969733,    -5.443857,    -1.039260>,
             <0.283443,    -3.162564,    -0.796536>,
             <0.764219,    -0.396339,    -0.596452>,
            <-5.818507,    -5.148710,     1.725471>,
            <-3.046107,    -5.633643,     1.768111>,
             <0.207069,    -3.352351,     2.010834>,
             <0.687845,    -0.586125,     2.210919>,
            <-7.121402,    -8.133587,     3.127131>,
             <2.579052,   -10.950022,     3.233690>,
             <4.089041,    -5.230404,     3.897720>,
             <6.646496,     0.560038,     6.933765>
                        }
```

```
define  b23    object {
                  bezier  2, 0.05,  3,  3,
       <3.010568,    -2.939457,    -12.403823>,
       <5.400538,     0.895668,    -12.403738>,
       <4.485245,     4.545848,    -12.846294>,
       <0.519430,     8.286380,    -12.888933>,
       <0.764219,    -0.396339,     -0.596452>,
       <1.244994,     2.369887,     -0.396368>,
      <-1.046631,     5.621045,     -0.238923>,
      <-3.819031,     6.105978,     -0.281563>,
       <0.687845,    -0.586125,      2.210919>,
       <1.168620,     2.180100,      2.411003>,
      <-1.123004,     5.431259,      2.568448>,
      <-3.895405,     5.916192,      2.525808>,
       <6.646496,     0.560038,      6.933765>,
       <9.203951,     6.350481,      9.969809>,
      <-0.198922,     8.480721,      5.740788>,
      <-6.165601,    10.466374,      6.160175>
                }

define  b24    object {
                  bezier  2, 0.05,  3,  3,
       <0.519430,     8.286380,    -12.888933>,
      <-3.446385,    12.026911,    -12.931573>,
     <-10.319574,     9.855964,    -13.274297>,
     <-13.157171,     3.747712,    -13.266072>,
      <-3.819031,     6.105978,     -0.281563>,
      <-6.591431,     6.590911,     -0.324202>,
      <-9.844608,     4.309619,     -0.566926>,
     <-10.325383,     1.543393,     -0.767010>,
      <-3.895405,     5.916192,      2.525808>,
      <-6.667805,     6.401125,      2.483169>,
     <-10.316352,     3.854265,      1.274445>,
     <-10.599442,     1.220823,      1.557361>,
      <-6.165601,    10.466374,      6.160175>,
     <-12.132281,     6.579563>,
     <-14.247386,     5.453698,      6.759902>,
     <-14.989909,     2.110008,      5.359019>
                }


define CylinderBezier object {
  b21 + b22 + b23 + b24
}


object {
    SheetBezier
    scale <2,2,2>
    translate <-14,0,0>
    texture { checker Stone13, matte_black}
}

object {
     CylinderBezier
     rotate<-90,-45,0>
     translate < 15,0,0>
     Stone6
}
```



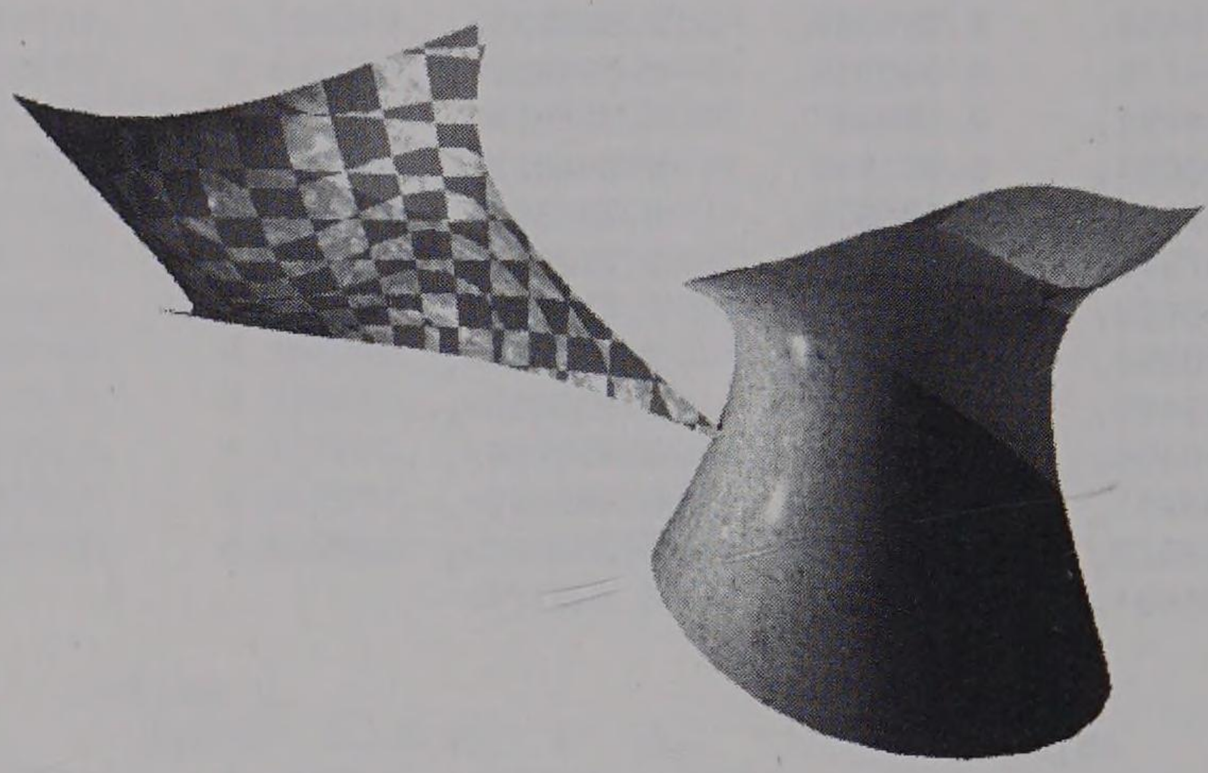


Figure 3.16
Flat and cylindrical Bézier patches (BEZIER.PI).

Bézier patches can become very complex to handle and normally require an external program to create and manipulate them (POVCAD, included with the book can handle Bézier patches).

Miscellaneous Objects

The following sections present a group of interesting objects and surface-generation techniques. In them, you work with blobs (objects that interact through attraction and repulsion) and with height fields to create landscapes. Finally, surface definition through the use of mathematical functions is discussed.

Blobs

Blobs are objects that can interact with other objects through potential fields (similar to a force field surrounding the object). They can attract and repel each other. In the process, they can be deformed or appear to merge with one another. There are three types of objects you can use in a blob: planes, cylinders, and spheres. The syntax is

```
blob threshold:
     component #1,
     component #2,
     ... etc ...
```

in which `threshold` is the minimum potential field value to be considered when computing the interaction between components and `component` is one of the following:

```
sphere <x, y, z> , strength, radius
cylinder <x0, y0, z0>, <x1, y1, z1>, strength, radius
plane <nx, ny, nz>, d, strength, distance
```

In all of the preceding component lines, `<x, y, z>` is the center of the blob sphere, `<x0, y0, z0>` and `<x1, y1, z1>` are the endpoints of the cylinder, `radius` is the radius of the sphere of the cylinder, `strength` is the strength of the potential field around the center of the component, `<nx, ny, nz>` is the normal vector of the plane, and `d` is the offset distance from the plane to origin along the normal.

The attractive or repulsive properties of a given component are based on the sign of the strength value. If the strength value is positive, objects attract. If the value is negative, they repel. There is no interaction between blobs, only between components of a blob.

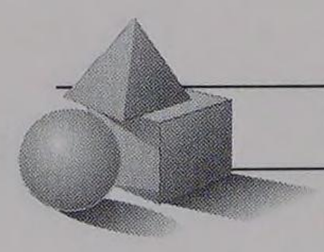
Plane components are infinite. To limit their size, use the `bounding_box < corner1>, < corner2>` statement. This box clips any part of the object that exceeds the `bounding_box` limits.

Listing 3.20 shows three blob objects with different behaviors. The first object (see fig. 3.17) is a dumbbell created with two attracting spheres. The object in the center is a plane being deformed by a sphere. Observe the effect of the `bounding_box` statement. I suggest you comment it out and ray trace it to see what happens. The final object is a cylinder and a sphere attracting each other. Observe how the cylinder deforms and merges with the sphere. Blob cylinders have hemispherical caps at the ends.

Listing 3.20. Using blobs (BLOB.PI).

```
// BLOB.PI

include "colors.inc"

define RedFinish  texture {
      surface {
         color red
         ambient 0.2
```

continues



Listing 3.20. Continued.

```
            diffuse 0.8
            specular white, 1
            microfacet Phong 5
            }
        }

viewpoint {
   from  <0, 10, -10>
   up    <0, 1, 0>
   at    <0, 1, 0>
   angle 35
   resolution 160, 160
   aspect 1.0
   }
// LIGHT_SOURCE
light < 10, 30, -20>

background midnightblue


object {
    object {
      blob 0.5:
      plane <0,1,0>, 0, 2, 5,
      sphere <0,3,0>, -4, 2
      RedFinish
    }
    bounding_box <-3,1,-1>, <3,3,3>  // <- bounding box
    scale <0.5,0.5,0.5>
}

object {
 object {
      blob 0.5:
      sphere <-2, 1.5, 0>, 1, 3,
      sphere < 2, 1.5, 0>, 1, 3
      reflective_yellow
}
 scale <0.4, 0.4, 0.4>
 translate <0, 3.7, 0>
}


object {
 object {
      blob 0.5:
      sphere <0, 0, 0>, 1, 3,
      cylinder < -2, 2, 0>, <2, 2, 0>, 1, 1.5
      reflective_yellow
}
 scale <0.6, 0.6, 0.6>
 translate <0, -3.5, 0>
}
```

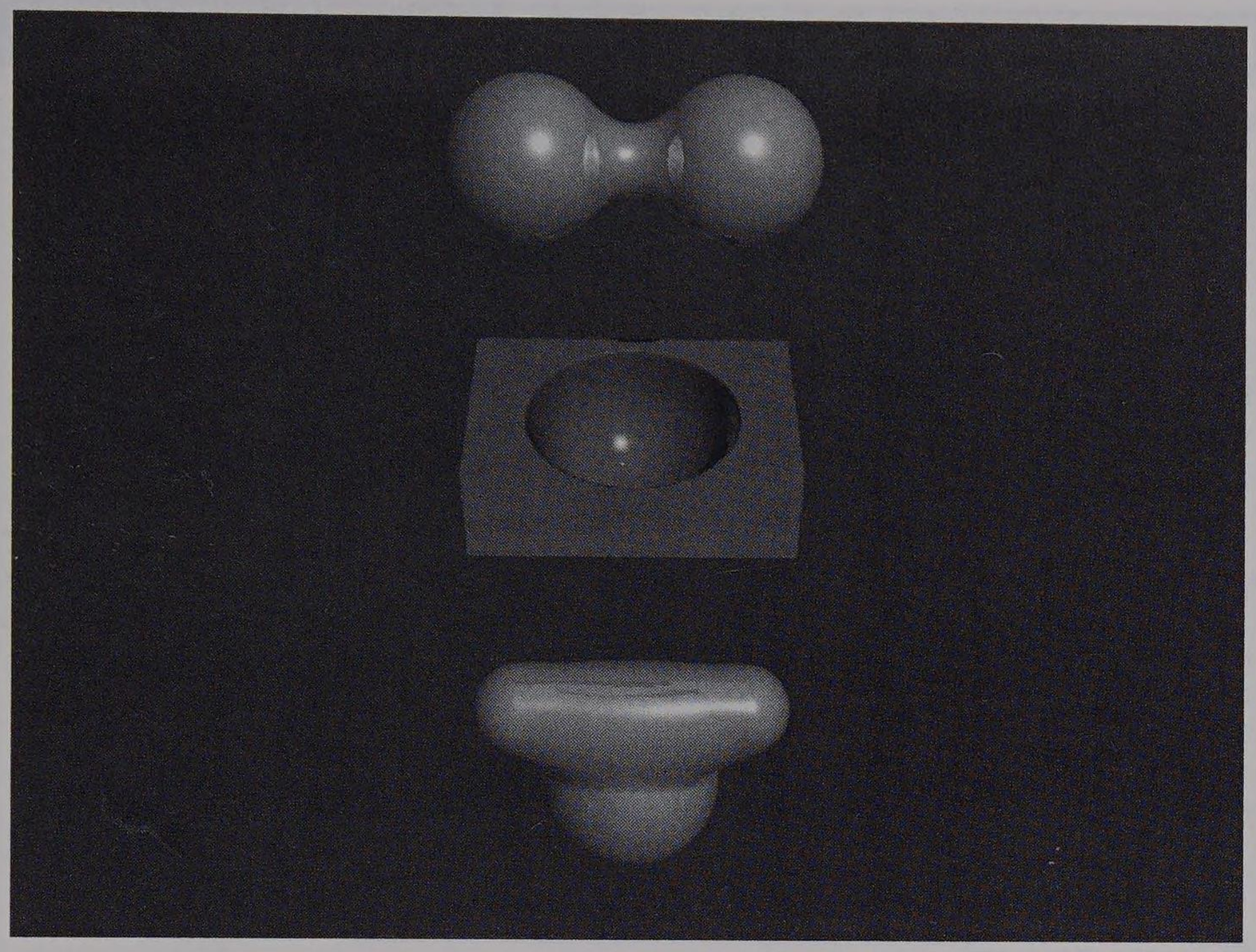


Figure 3.17
Three examples of blobs (BLOB.PI).

Height Fields

Height fields are commonly used to produce interesting and realistic landscapes. Landscapes are created by reading an image file. This file can be of any image as long as it is in Targa uncompressed format (8-, 16-, or 24-bit). The best images to use are those of plasma images like the ones created with Fractint. (See the introduction of this book for bulletin board phone numbers from which you can download Fractint and ray tracing utilities.)

The image file is read and the colors are converted into height values. Initially, the height field object is scaled to $1 \times 1 \times 1$ units, with the bottom left corner located at the origin. The syntax, which follows, is extremely simple for such a powerful feature.

```
height_field "filename"
```

or

```
smooth_height_field "filename"
```

If you use the `smooth_height_field` option, the final ray-traced image is smoother at the expense of more memory. Listing 3.21 shows a height field generated with a plasma image. Figure 3.18 shows the original plasma image, and figure 3.19 shows the plasma image converted into a height field. A stone texture was attached to it.

Listing 3.21. Using height fields (HEIGHT.PI).

```
// HEIGHT.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

define Plane_XZ  object {
   polygon 4,  <-10000,0,10000>, <-10000,0,-10000>,
               <10000,0,-10000>, <10000,0,10000>
               }


viewpoint {
   from < 0, 10, -70>
   at <0,-10,0>
   up <0,1,0>
   angle 45
   resolution 100, 100
   aspect 1.0

   }

light <-45, 25, -4>
light <0, 25, 0>

object {
   Plane_XZ
   rotate <90,0,0>
   translate <0,0,10000>
   texture { surface {color White} }
}

object {
   height_field "plasma.tga"
   translate <-0.5, 0,-0.5>
   scale <100, 0.05, 100>
   rotate <0, -15, 0>
   Stone6
   }

object {
   Plane_XZ
   translate <0,-6.7, 0>
   texture { surface {color Blue} }
}
```

In addition to the height field, there are two planes being used. The first plane used in the image creates a white sky in the background. The second plane is used to create the effect of an ocean. You can experiment by modifying the height of the ocean in the y axis to hide or show the height field more.

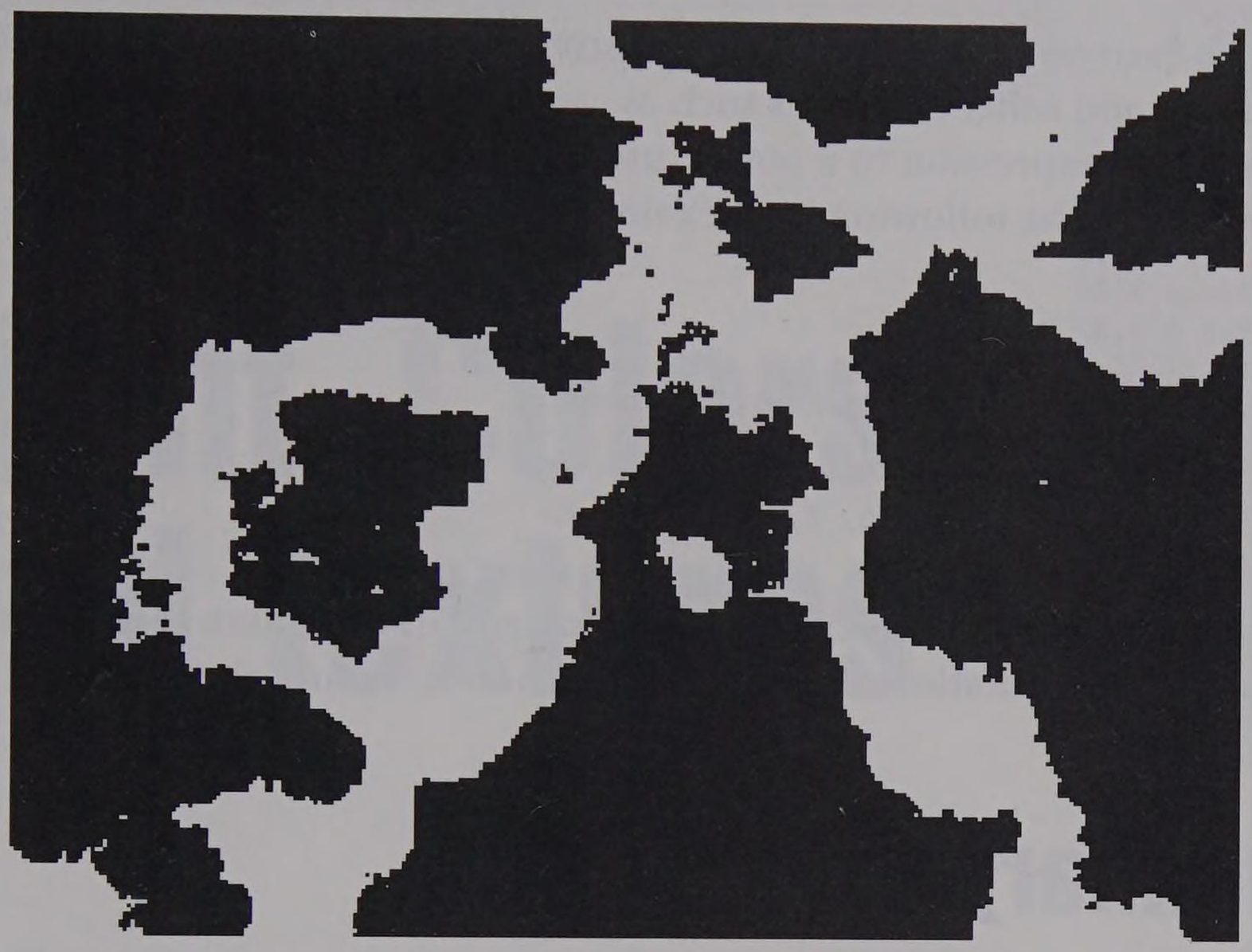


Figure 3.18
A plasma image (PLASMA.TGA).

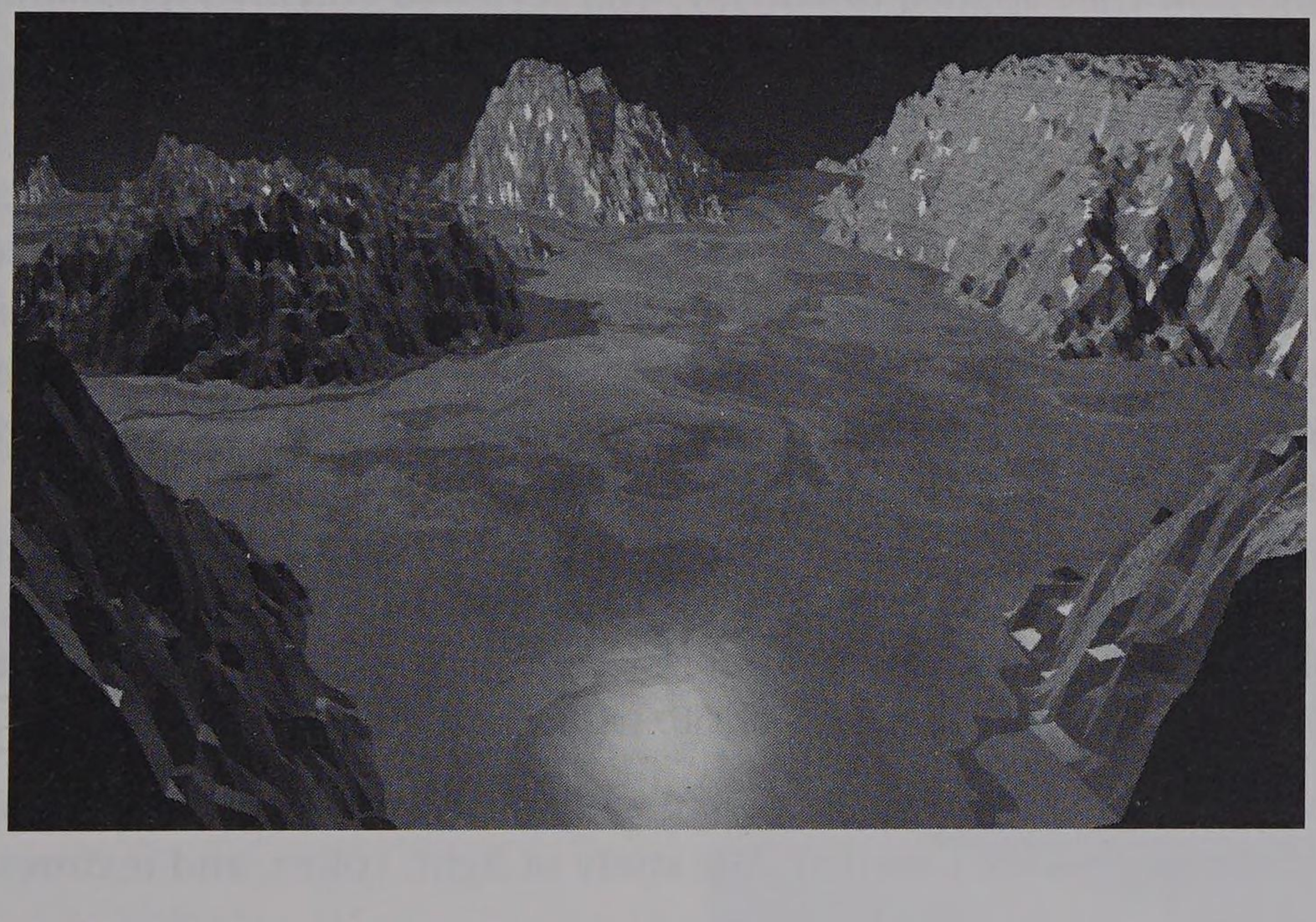


Figure 3.19
A plasma image as a height field (HEIGHT.PI).

Functions

Another method of creating surfaces is through the definition of mathematical functions. The syntax for this method is

```
function mathematical function
```

in which `"mathematical function"` can be any relationship between the variables `x`, `y`, `z` and other functions such as `cos()`, `sin()`, `tan()`, `exp()`, and others. To elevate an expression to a power, use the `^` sign (for example, `3^2` means *three squared*). The following is an example of a function:

```
define a 2.0
define b 3.0
object {
        // this is an elliptical cylinder
        function (x^2)/a^2 + (y^2)/b^2 - 1
        shiny_red
        bounding_box <-5,-5,-5>, <5, 5, 5>
}
```

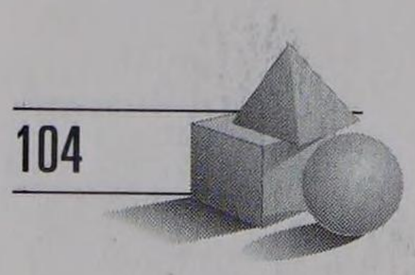
Due to numerical inaccuracies, the final image can have dark spots. Check Polyray's documentation for techniques of how to reduce this problem.

Summary

In this chapter, you covered most of the same concepts presented in Chapter 1 and how they applied to Polyray. You reviewed Polyray's left-handed coordinate system and how to rotate, translate, and scale objects in it. You studied viewpoint definition and parameters.

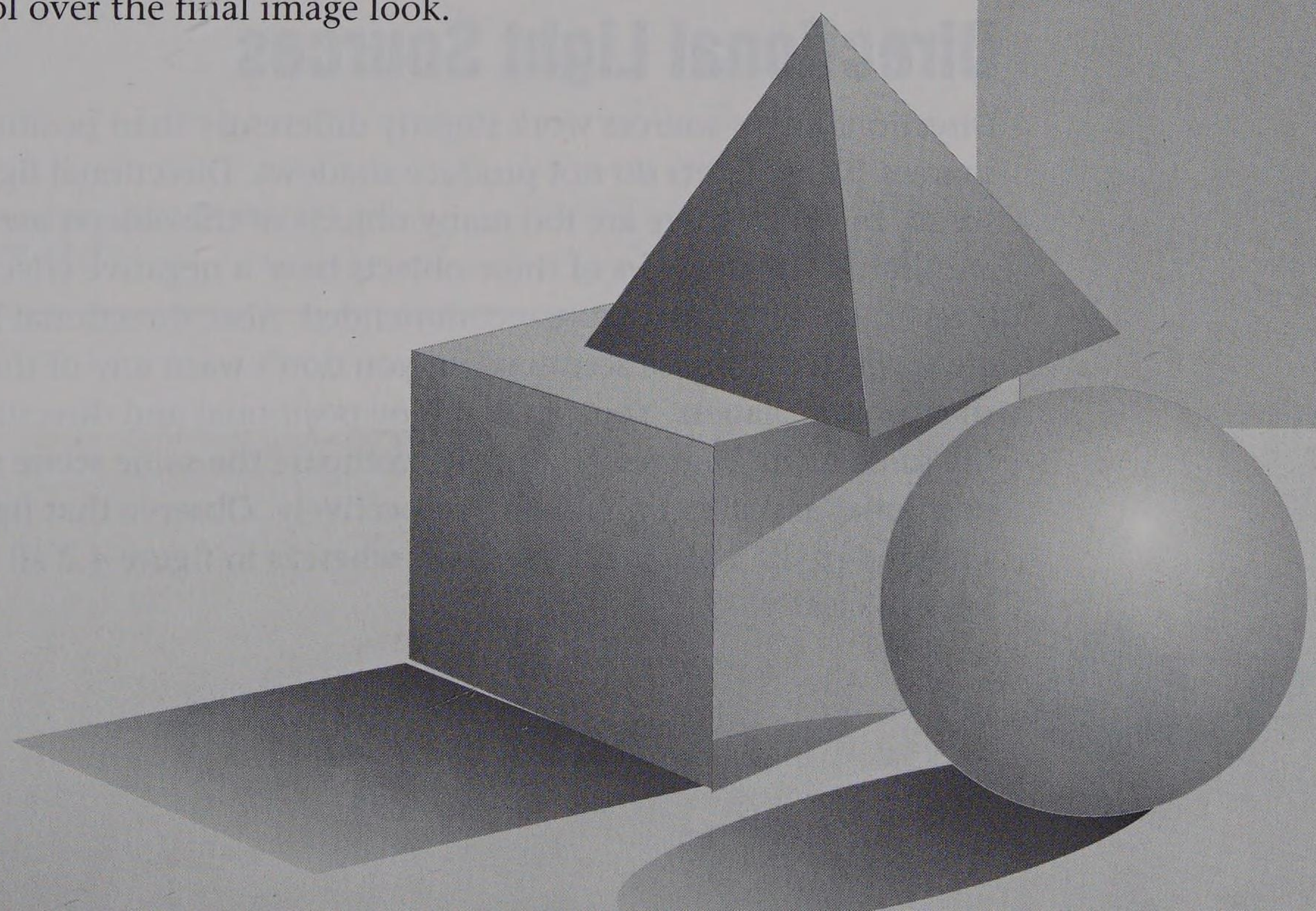
You also saw an array of Polyray primitives such as boxes, planes, spheres, discs, ellipsoids, and the torus. The task of modeling surfaces by using small utility programs was discussed and examples of programs that create flat and smooth triangles were studied. Other surfaces such as polygons, lathe, sweep, and Bézier were created and the degree of difficulty when developing them was discussed. Finally, you were exposed to some strange but interesting objects such as blobs, whose components interact with each other and height fields, commonly used to create landscapes. Surface generation by the use of mathematical functions was discussed briefly; however, it requires some math knowledge of functions and methods to reduce numerical inaccuracies.

In the next chapter, you start studying what makes one surface look like a mirror and another look dented. The study of light, colors, and textures is next.



Light, Colors, and Textures

In this chapter, you study three key elements to photo-realistic image creation. First, the chapter covers different types of light sources, ranging from the standard positional type to spot lights. Next, you learn how to apply colors to surfaces and the background of a scene, and even how to create haze on an image. Finally, you study the topic of textures in detail. Textures have a wide variety of parameters that enable you to create different types of finishes on your objects. By understanding the relationship between the parameters, you gain control over the final image look.



Light Sources

Light sources not only help you see the objects that compose a scene, but they also let you create shadows and the "mood" of the scene. There are three types of light sources available: positional, directional, and spot light. The simplest and most commonly used light source is the positional light source.

Positional Light Sources

Positional light sources can be compared to a light bulb. To use one in your scene, you need to define the location `<x,y,z>` in space and color. The syntax is

```
light color, <x, y, z>
```

or

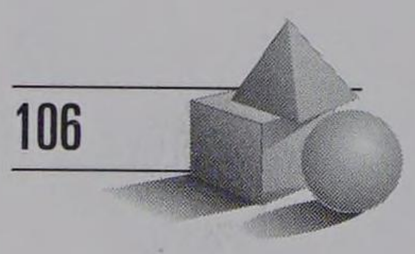
```
light <x,y,z>
```

in which *`color`* is either a color vector of the type <Red, Green, Blue> or a token word such as `white` or `yellow`. The values for Red, Green, and Blue normally vary from 0 to 1.

In the second format, `light <x,y,z>`, because no color is specified, `white` light is assumed. It is useful to know the token names for the available colors. The next section contains a list of the available color names. All the examples you have seen so far in this book have used positional light sources.

Directional Light Sources

Directional light sources work slightly differently than positional light sources. These lights do not produce shadows. Directional light is handy in scenes in which there are too many objects or the objects are too near one another. If the shadows of these objects have a negative effect on the scene, directional light sources are recommended. Also, directional light sources are suggested in space scenes when you don't want any of the objects to cast shadows on planets. You can combine positional and directional light in the same scene. Figures 4.1 and 4.2 compare the same scene ray-traced with positional and directional light, respectively. Observe that figure 4.1 has shadows of the objects on the floor, whereas in figure 4.2 all the objects can be seen clearly.



Listing 4.1. Using positional light sources (LIGHT.PI).

```
//LIGHT.PI

include "texture.inc"
include "colors.inc"

define Plane_XZ  object {
              polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                         <10000,0,-10000>, <10000,0,10000>
                 }

viewpoint {
   from <0,0,-10>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 100, 100
   aspect 1.0
   }

background <0, 0, 0>

light <-10,6,-20>
light < 10,6,-20>

object {
   sphere <0, 0, 0>, 2
   shiny_red
   }

object {
   sphere <3, 0, 3>, 2
   shiny_yellow
   }

object {
   sphere <-3, 0, 3>, 2
   shiny_green
   }

object {
   Plane_XZ
   texture { surface {color White}}
   translate <0,-3,0>
}
```

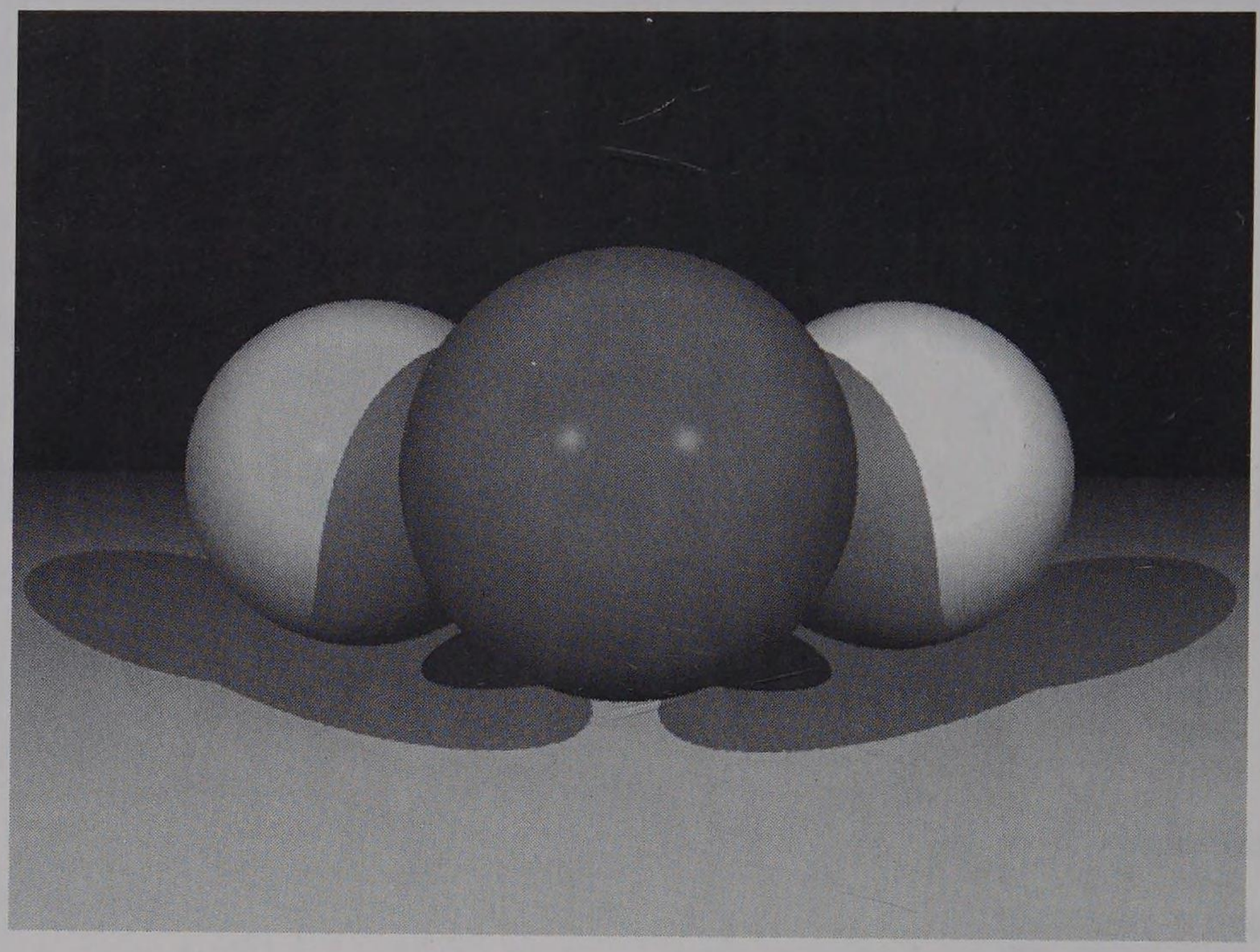


Figure 4.1

Spheres illuminated with positional light sources (LIGHT.PI).

Listing 4.2. Using directional light sources (DIRECTLT.PI).

```
//DIRECTLT.PI
include "texture.inc"
include "colors.inc"

define Plane_XZ  object {
              polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                         <10000,0,-10000>, <10000,0,10000>
                 }

viewpoint {
   from <0,0,-10>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 100, 100
   aspect 1.0
   }

background <0, 0, 0>

directional_light <-10,6, -20>
directional_light < 10,6, -20>

object {
   sphere <0, 0, 0>, 2
   shiny_red
   }
```

```
object {
   sphere <3, 0, 3>, 2
   shiny_yellow
   }

object {
   sphere <-3, 0, 3>, 2
   shiny_green
   }

object {
   Plane_XZ
   texture { surface {color White}}
   translate <0,-3,0>
}
```

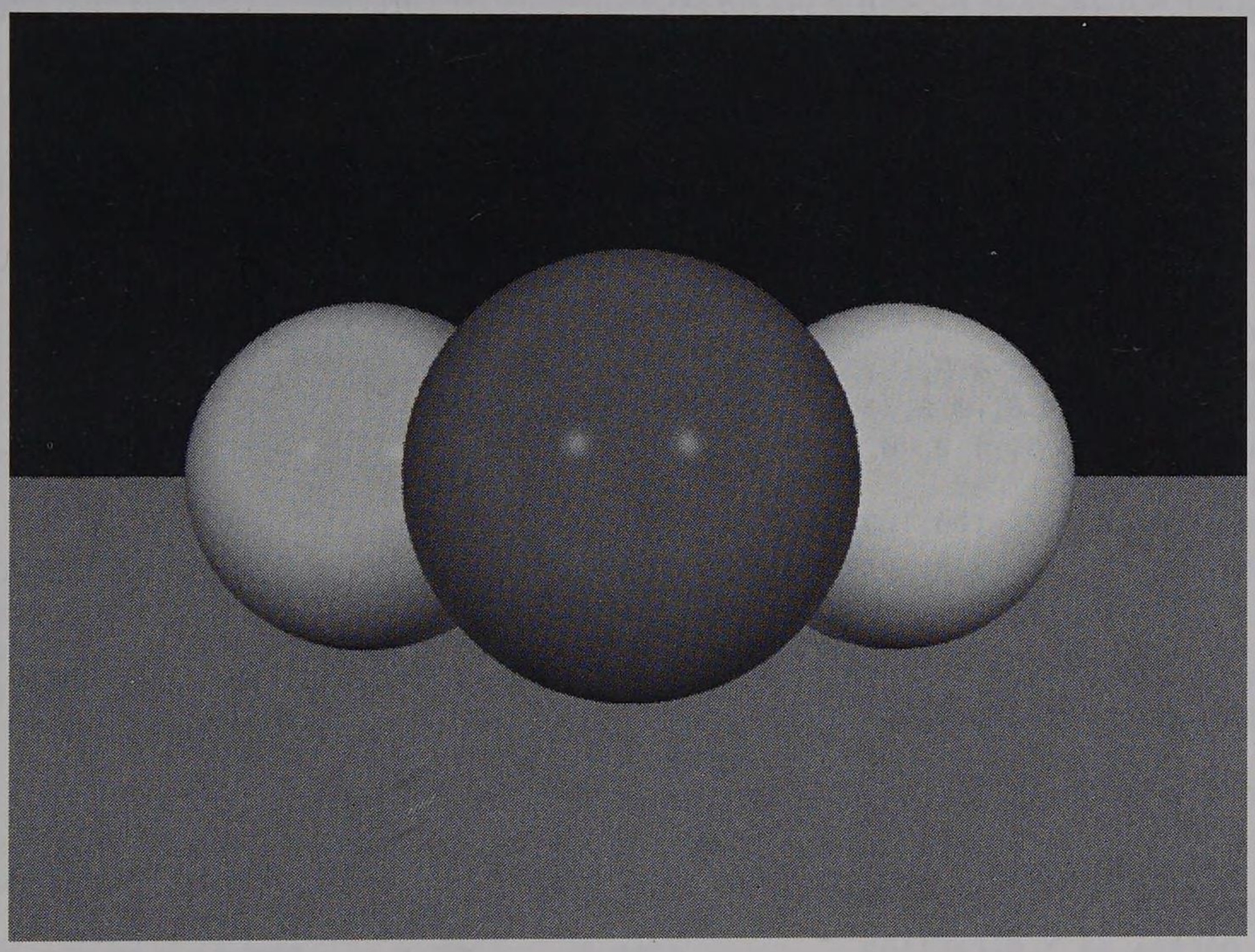


Figure 4.2
Spheres illuminated with directional light sources (DIRECTLT.PI).

Spot Light Sources

Spot lights are basically cones of light that you can use to create some interesting effects. The walls of the cone of light are invisible; only the base, or "hot spot," is visible. The syntax is

```
spot_light color, <location>, <point at>, Tightness, Ang, Fall-off
```

or

```
spot_light color, <location>, <point at>
```

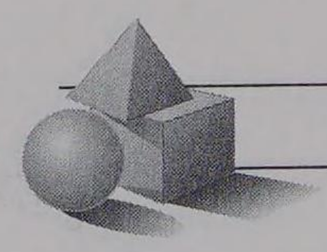
In which the placeholders stand for the following: *color* indicates the color of the spot light. Use <R,G,B> vector or a token name. *<location>* is replaced with the location of the apex of the cone of light. *<point at>* indicates the location at which the spotlight is aiming. *Tightness* determines the shape of the spot (the smaller the number, the larger the spot). A good value to start with for *Tightness* is 3 (change as necessary; it requires some experimentation). *Ang* specifies the angle in degrees, where the spot is in full effect. *Fall-off* indicates an angle larger than *Ang* where the spot light turns off.

Listing 4.3 shows an example of using spot lights.

Listing 4.3. Using spot lights (SPOTLITE.PI).

```
//SPOTLITE.PI

include "colors.inc"
include "texture.inc"

define LampColor reflective_grey

viewpoint {
 from      <0.0, 0.9, -2.8>
 at        <-0.6, 0.2, 0.0>
 up        <0.0, 1.0, 0.0>
 angle          45
 resolution     100,100
 aspect         1.0
}

spot_light White , <-0.28,0.67, 0>, <-1.5, 0, 0>, 3, 45, 50
light <0, 100, -100>

background Grey

// PLANE (Polygon)
object {
// normal in Y direction
   polygon 4, <-1,0,1>, <-1,0,-1>, <1,0,-1>, <1,0,1>
   scale <3,1,3>
   texture {
       checker matte_white, matte_black
       scale <0.1, 1, 0.1>
   }
}

// LAMP BASE
object {
// SPHERE
object {
    sphere < 0, -0.1, 0>, 0.3
    LampColor
```

```
}
-
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <0.5, 0.5, 0.5>
    translate <0, -0.5, 0>
}
   LampColor
}

// LAMP "NECK"
object  {
// TORUS
object {
    torus  0.4, 0.05, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <90, 0, 0>
    translate <-0.375, 0.3, 0>
}
*
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <0.4, 0.4, 0.4>
    translate <0.125, 0.575, 0>

}
 LampColor
}

// OBJECT ON FLOOR
object {
   sphere <-1.3,0.1, 0>, 0.1
   shiny_red
}

// LAMP "HEAD" DONE USING POVCAD AND LATHE.BAS
object {
   object {
    lathe  2 , <0, 1, 0>,  9 ,
          <    0.000,    0.575,    0.000>,
          <    0.075,    0.550,    0.000>,
          <    0.143,    0.488,    0.000>,
          <    0.155,    0.410,    0.000>,
          <    0.185,    0.332,    0.000>,
          <    0.223,    0.284,    0.000>,
          <    0.311,    0.215,    0.000>,
          <    0.374,    0.119,    0.000>,
          <    0.408,    0.011,    0.000>
     LampColor
   }
   rotate <0, 0, -65>
   translate <-0.6, 0.675, 0>
}
```



The spot light is located inside the lamp, where the bulb would be in a real lamp. The point aimed at is near the sphere (see fig. 4.3).

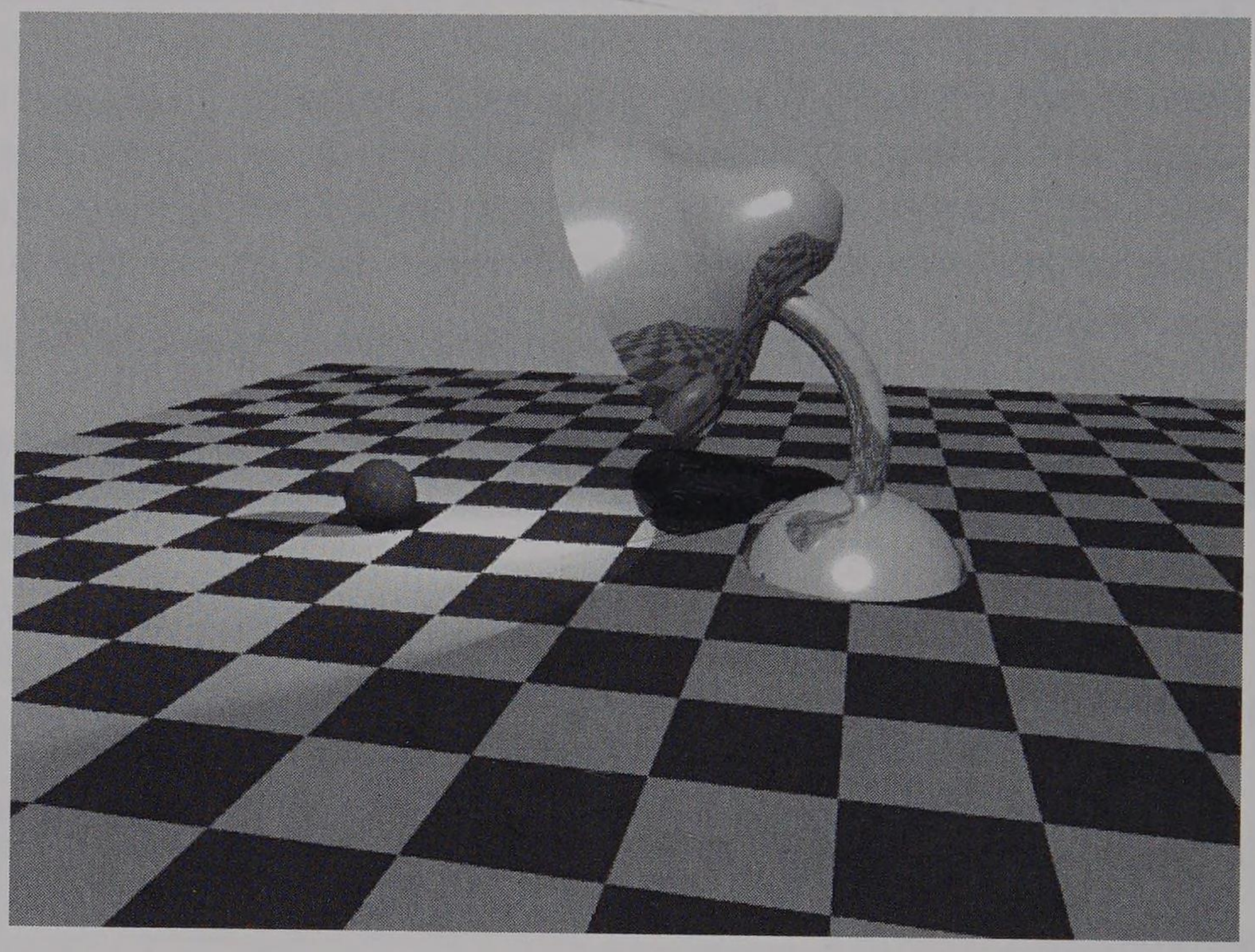


Figure 4.3
Using a spot light to model a lamp (SPOTLITE.PI).

Figure 4.3 has some interesting features that we should discuss further:

- The lamp's color is defined by means of the token name `LampColor`. This enables you to modify the color of the lamp without having to edit each individual component.
- A spot light and a positional light source are used in the same scene. The positional light source is used to aid the spot light and create a slightly lit scene.
- In the definition of the plane, the size of the checkered tiles was modified by scaling inside the texture definition:

```
texture {  checker matte_white, matte_black
           scale <0.1, 1, 0.1>
}
```

This is done to force the tiles to be smaller without affecting the size of the plane.

- The base and neck of the lamp were created using Boolean operations. The base is a sphere that had a box subtracted from it to create a hemisphere. The neck is created by intersecting a torus and a box. In other words, whatever part of the torus is inside the box will remain. The rest is discarded.
- The lamp head was created in POVCAD by drawing the contour. The data was saved and the program LATHE.BAS was used. Observe that a type 2 (smooth) lathe object is used to give better quality.

Colors

Colors in Polyray can be defined in two ways:

- By using a vector of the form <R, G, B>, in which R, G, and B stand for the red, green, and blue components of the color. For example, <0,0,0> is black, and <1,1,1> is white.
- By using the defined token name of the color. Instead of using the color <0,0,0>, it is better to use the token name `black`. The file COLORS.DOC included in Polyray has a list of the token names of all the colors and their respective <R,G,B> values.

NOTE: The predefined colors (like the ones included in COLOR.DOC) can be either uppercase or lowercase: both Black and black are OK. If you create your own colors using the define statement, those colors *are* case-sensitive—if you create `define MYCOLOR <0.5, 0.2,0.45>`, then you have to use the name MYCOLOR exactly as you defined it.

There are a number of named colors acceptable to Polyray. These were present in MTV and are carried through to Polyray. The colors (case is important) and their RGB values are shown in Table 4.1.

Table 4.1. Colors defined in Polyray (COLORS.DOC).

Name	Red	Green	Blue
Aquamarine	0.439216	0.858824	0.576471
Black	0	0	0
Blue	0	0	1
BlueViolet	0.623529	0.372549	0.623529
Brown	0.647059	0.164706	0.164706
CadetBlue	0.372549	0.623529	0.623529
Coral	1	0.498039	0
CornflowerBlue	0.258824	0.258824	0.435294
Cyan	0	1	1
DarkGreen	0.184314	0.309804	0.184314
DarkOliveGreen	0.309804	0.309804	0.184314
DarkOrchid	0.6	0.196078	0.8
DarkSlateBlue	0.419608	0.137255	0.556863
DarkSlateGray	0.184314	0.309804	0.309804
DarkSlateGrey	0.184314	0.309804	0.309804
DarkTurquoise	0.439216	0.576471	0.858824
DimGray	0.329412	0.329412	0.329412
DimGrey	0.329412	0.329412	0.329412
Firebrick	0.556863	0.137255	0.137255
ForestGreen	0.137255	0.556863	0.137255
Gold	0.8	0.498039	0.196078
Goldenrod	0.858824	0.858824	0.439216
Gray	0.752941	0.752941	0.752941
Green	0	1	0
GreenYellow	0.576471	0.858824	0.439216
Grey	0.752941	0.752941	0.752941
IndianRed	0.309804	0.184314	0.184314

Name	Red	Green	Blue
Khaki	0.623529	0.623529	0.372549
LightBlue	0.74902	0.847059	0.847059
LightGray	0.658824	0.658824	0.658824
LightGrey	0.658824	0.658824	0.658824
LightSteelBlue	0.560784	0.560784	0.737255
LimeGreen	0.196078	0.8	0.196078
Magenta	1	0	1
Maroon	0.556863	0.137255	0.419608
MediumAquamarine	0.196078	0.8	0.6
MediumBlue	0.196078	0.196078	0.8
MediumForestGreen	0.419608	0.556863	0.137255
MediumGoldenrod	0.917647	0.917647	0.678431
MediumOrchid	0.576471	0.439216	0.858824
MediumSeaGreen	0.258824	0.435294	0.258824
MediumSlateBlue	0.498039	0	1
MediumSpringGreen	0.498039	1	0
MediumTurquoise	0.439216	0.858824	0.858824
MediumVioletRed	0.858824	0.439216	0.576471
MidnightBlue	0.184314	0.184314	0.309804
Navy	0.137255	0.137255	0.556863
NavyBlue	0.137255	0.137255	0.556863
Orange	0.8	0.196078	0.196078
OrangeRed	1	0	0.498039
Orchid	0.858824	0.439216	0.858824
PaleGreen	0.560784	0.737255	0.560784
Pink	0.737255	0.560784	0.560784
Plum	0.917647	0.678431	0.917647

continues

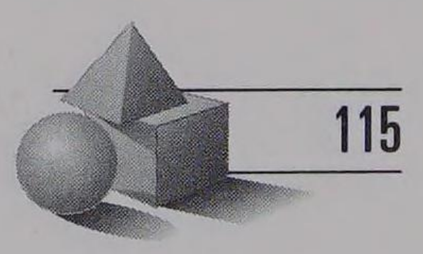


Table 4.1. Continued.

Name	Red	Green	Blue
Red	1	0	0
Salmon	0.435294	0.258824	0.258824
SeaGreen	0.137255	0.556863	0.419608
Sienna	0.556863	0.419608	0.137255
SkyBlue	0.196078	0.6	0.8
SlateBlue	0	0.498039	1
SpringGreen	0	1	0.498039
SteelBlue	0.137255	0.419608	0.556863
Tan	0.858824	0.576471	0.439216
Thistle	0.847059	0.74902	0.847059
Turquoise	0.678431	0.917647	0.917647
Violet	0.309804	0.184314	0.309804
VioletRed	0.8	0.196078	0.6
Wheat	0.847059	0.847059	0.74902
White	0.988235	0.988235	0.988235
Yellow	1	1	0
YellowGreen	0.6	0.8	0.196078

You can use colors on objects, on the background, and when you're creating a haze effect.

Surface

Using colors on an object's surface is straightforward. To attach a color to the surface, use the following syntax:

```
texture { surface { color MyColor } }
```

An example of a red sphere is

```
object {
      sphere <0,0,0>, 2
      texture { surface { color Red } }
}
```

where `Red` is a color found in Listing 4.3. The keyword `texture` was used to define the color of the surface. This keyword and its components are explained in the "Textures" section later in this chapter. There are two different methods you can use to specify the color of an object inside a texture definition:

Method 1:

```
object {
   sphere <0,0,0>, 2
   texture { surface { color Red } }
}
```

Method 2:

```
define MyColorRed texture { surface { color Red } }

      object {
          sphere <0,0,0>, 2
          MyColorRed
      }
```

Method 1 is the long format, where all the details of the texture definition are written inside of the object. Method 2 uses the `a define` statement to attach a token name to the texture definition as a whole. Method 2 is recommended.

Background

When Polyray finds out that a light ray doesn't intersect any of the objects in a scene, the pixel on-screen is set to black as the default. You can change the color of the background by using the keyword `background` as in the following:

```
background color
```

in which `color` is either an <R,G,B> vector or a token name.

You can also use an image file as a background. The syntax for this operation is background planar_imagemap (image("*file.tga*"),P) in which the Targa image file *file.tga* is projected to the background and scaled to fit in the background frame. Listing 4.4 shows the use of a `planar_imagemap` as a background (see fig. 4.4 for the output).

Listing 4.4. Using an image file as a background (BACK.PI).

```
//BACK.PI

include "texture.inc"
include "colors.inc"

viewpoint {
   from <0,0,-10>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 100, 100
   aspect 1.0
   }

background planar_imagemap(image("plasma.tga"),P)

light <-10,6, -20>
light < 10,6, -20>

object {
   sphere <0, 0, 0>, 2
   shiny_red
   }

object {
   sphere <3, 0, 3>, 2
   shiny_blue
   }

object {
   sphere <-3, 0, 3>, 2
   shiny_green
   }
```

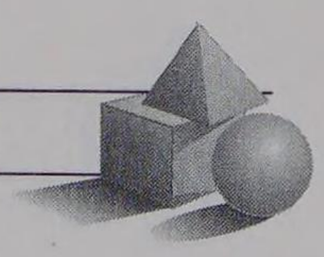
Listing 4.4 uses the same plasma image that created the height fields in Chapter 3. Figure 4.4 was ray traced without antialiasing.

Global Haze

Polyray creates a haze effect by using a color that becomes thicker the farther the light has to travel. The syntax for haze is

```
haze thickness, start, color
```

in which *thickness* is the amount of nontransparency of the haze. The value is between 0 and 1, with 0 being nontransparent and 1 being fully transparent. The parameter *start* determines at what distance the haze effect starts. The parameter *color* can be a token name or a color vector.



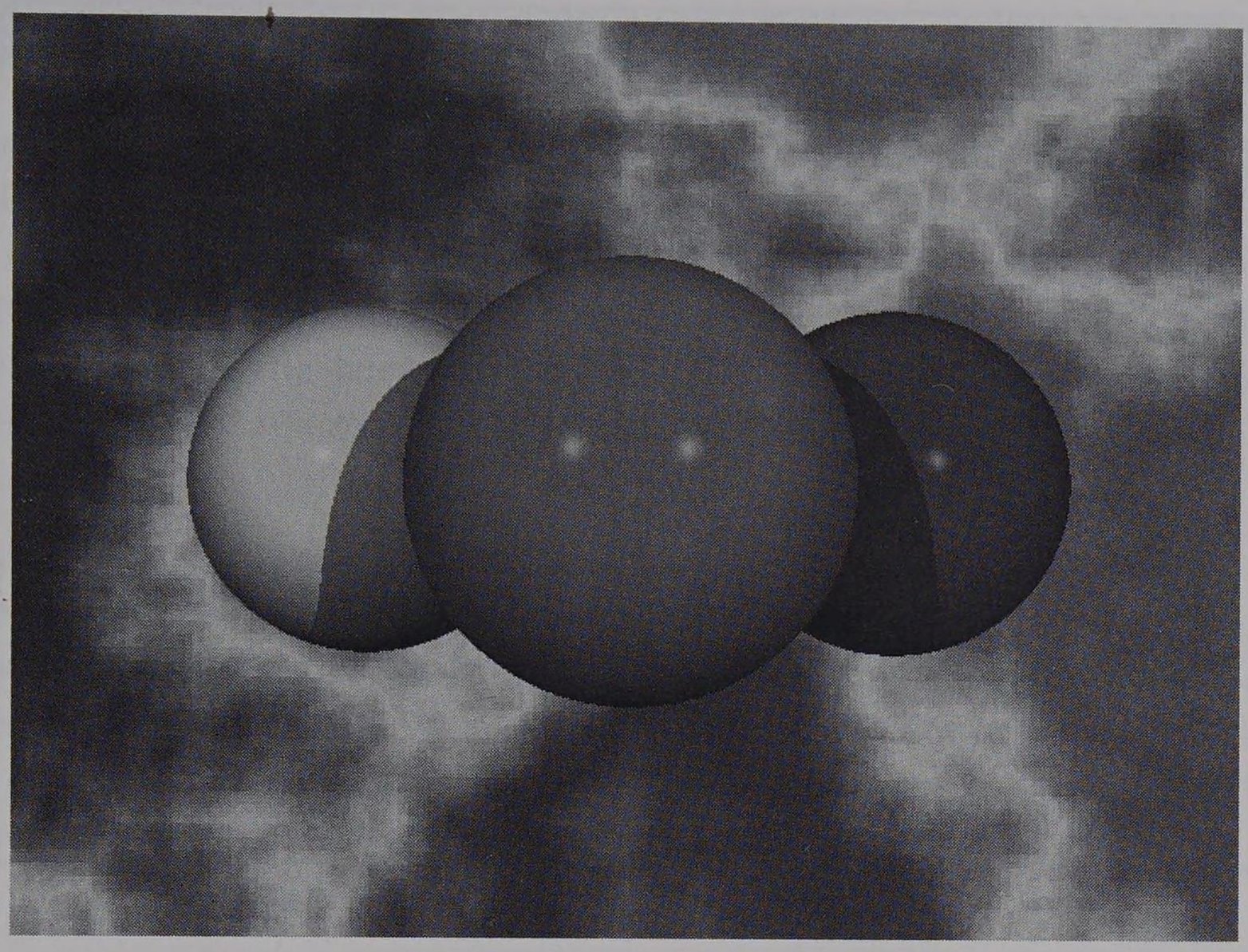


Figure 4.4

Using an image as a background (BACK.PI).

Listing 4.5 is a modification of HEIGHT.PI (see Listing 3.2). The statements

```
background Gray
```

and

```
haze 0.95, 50, Gray
```

were added. Usually, haze is the same color as the background.

Listing 4.5. A landscape with the haze effect (HAZE.PI).

```
// HAZE.PI

include "colors.inc"
include "texture.inc"
include "stones.inc"

define Plane_XZ  object {
            polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                       <10000,0,-10000>, <10000,0,10000>
          }


viewpoint {
   from < 0, 10, -70>
   at <0,-10,0>
   up <0,1,0>
   angle 45
   resolution 100, 100
   }
```

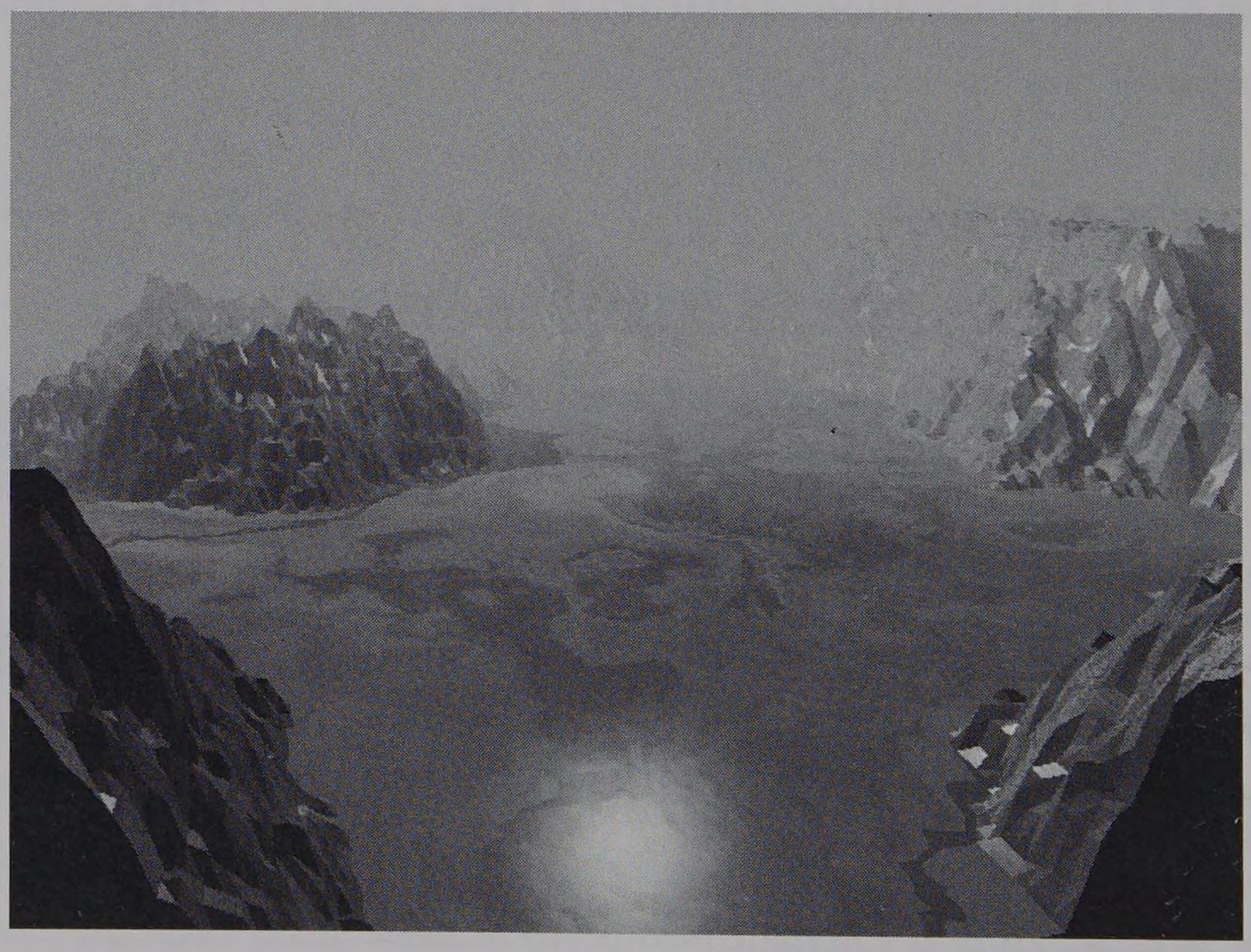
continues

Listing 4.5. Continued.

```
light <-45, 25, -4>
light <0, 25, 0>

background Gray
haze 0.95, 50, Gray

object {
   height_field "plasma.tga"
   translate <-0.5, 0,-0.5>
   scale <100, 0.05, 100>
   rotate <0, -15, 0>
   Stone6
   }

object {
   Plane_XZ
   translate <0,-6.7, 0>
   texture { surface {color Blue} }
}
```

Figure 4.5 shows the landscape immersed in haze.

Figure 4.5
Adding a haze effect to a landscape (HAZE.PI).



Textures

This section familiarizes you with Polyray's texture format. The format is flexible and has a variety of parameters that you can modify. If you feel comfortable using the standard colors or some of the previously used textures, it's time to go into more detail on how to create your own textures or modify the existing ones. Textures are challenging, and the rewards for understanding how to manipulate them are great.

Texture Surface Parameters

Polyray supports what is commonly called a *standard shading model*. A standard shading model describes how a surface handles incident light in terms of reflection, refraction, and diffusivity. Examples of standard surface textures look roughly like this:

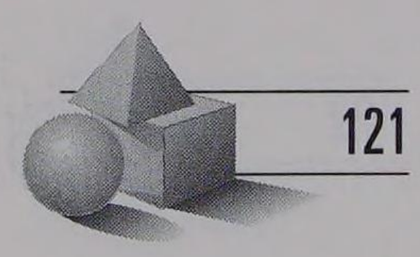
Reflective Red Surface

```
texture {
     surface {
          color red
          ambient 0.1
          diffuse 0.5
          specular white, 0.5
          microfacet Phong 7
          reflection white, 0.5
     }
}
```

Blue Glass Surface

```
texture {
     surface {
          color blue
          ambient 0.1
          diffuse 0
          specular white, 0.2
          reflection white, 0.1
          transmission white, 1, 1.5
     }
}
```

The next section explains the various parameters in detail.



Color

The keyword `color` is followed by an <R,G,B> vector or a token color name. This is the color of the surface. If this statement is included inside of the texture definition, it is not required to declare the color on the other parameters.

Ambient Light

Ambient light hits the object's surface from all directions. As a result, the object becomes visible. The syntax for creating ambient light is

```
ambient color, scale
```

or

```
ambient scale
```

In the first example, *color* is the color of the surface. You may want to use the second format if you already declared the color using the keyword `color`. For example,

```
color red
ambient 0.1
```

has the same effect as

```
ambient red, 0.1
```

You can use other colors to create different effects. The parameter *scale* determines the amount of contribution of the color to the scene. A value of 0 will show no color at all and a scale value of 1 will use the full color effect.

Diffuse Light

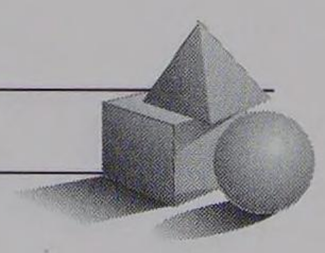
Diffuse light is light that has been absorbed by the surface and spread out equally in all directions (see fig. 4.6). The syntax for creating the diffuse light effect is

```
diffuse color, scale
```

or

```
diffuse scale
```

The *scale* parameter determines the amount of contribution of diffuse light to the scene. A value of 0 does not diffuse light, whereas a scale value of 1 diffuses all the incident light.



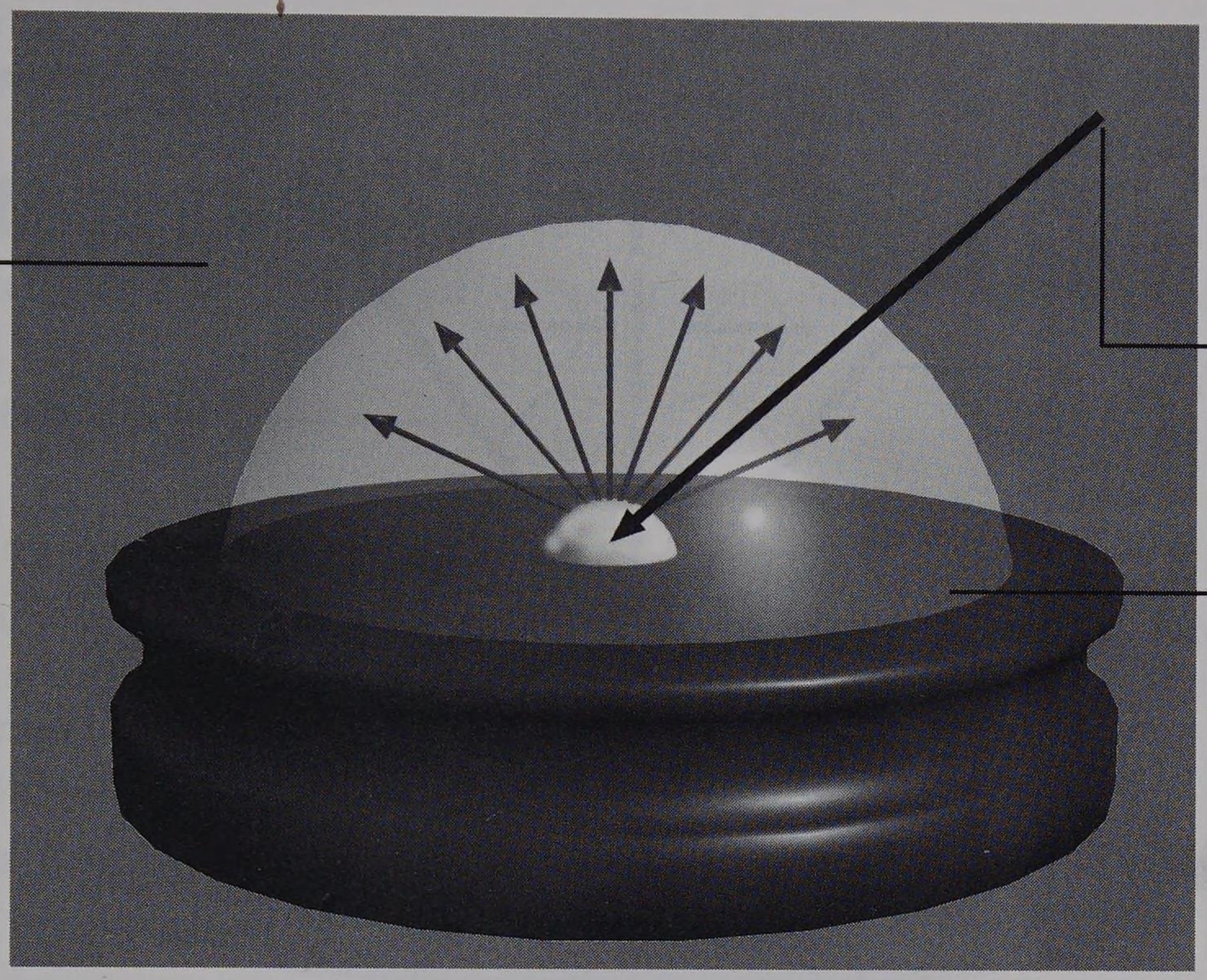


Figure 4.6
Diffuse reflection.

Specular Highlights

On a perfect mirror surface, the incident angle of the light is equal to the reflection angle (see fig. 4.7a). On a rough surface (see fig. 4.7b), the reflected light is concentrated on an area. What the observer sees is called a *specular highlight*. The light scatters or spreads out due to the roughness of the surface. The syntax for a specular highlight is

```
specular color, scale
```

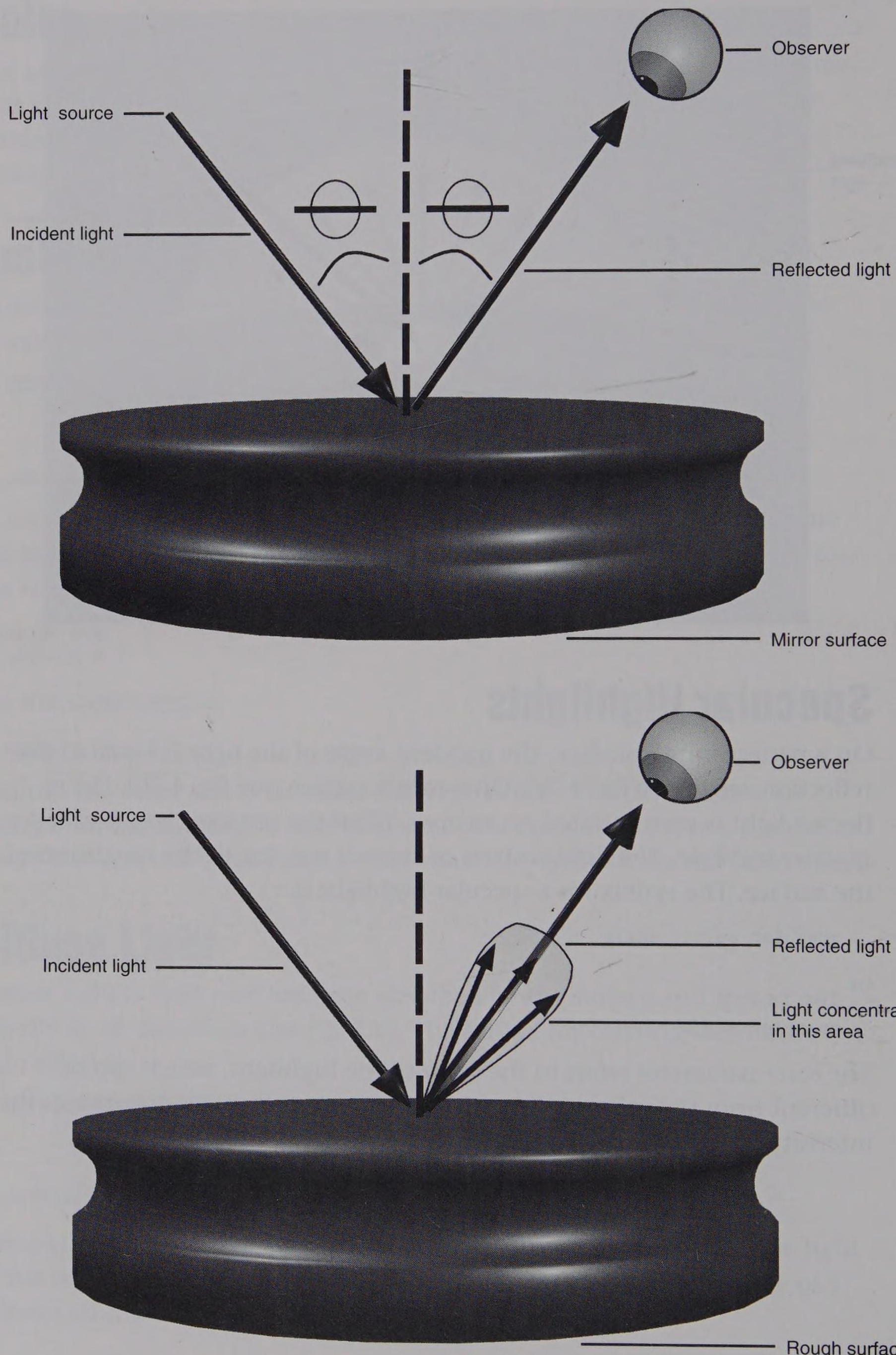
or

```
specular scale
```

The `color` parameter refers to the color of the highlight, which can be different from the color of the surface. The `scale` parameter determines the intensity of the specular highlight, the value being between 0 and 1.

Figure 4.7

The effect of surface roughness on reflected light.



Later in this chapter, the section "Microfacet Distribution" discusses how to experiment with different surface roughness models.

Reflected Light

Reflected light bounces off surfaces at an angle. The syntax for reflected light is

```
reflection color, scale
```

or

```
reflection scale
```

If you use the first format, the color becomes a filter. Normally, the second format is used because it reflects all colors. The `scale` value determines the contribution of the reflected light to the final color of the surface. The values are between 0 and 1.

As an example of different scale values for some surfaces, metallic surfaces have a scale of 0.25, reflective surfaces are 0.5, and mirrors 1.0.

Transmitted Light

On transparent materials, light goes through the surface and bends. (The same effect occurs when you put a spoon in a glass filled with water. The spoon's image bends.) The *index of refraction*, or IOR, determines how much the light will bend. Table 4.2 shows various IOR values for different materials.

Table 4.2. Index of refraction values for various materials.

Material	IOR
Diamond	2.42[2]
Ice	1.31[2]
Quartz	1.46[2]
Water	1.33[1,2]
Air	1.0003[1]
Crown Glass	1.52[1,2]
Rock Salt	1.54[2]

References:

1. "Physics, Part 2" by D. Halliday and R. Resnick, 3rd Ed. 1978, John Wiley and Sons.

2. "Fundamentals of Waves, Optics, and Modern Physics" by H. D. Young, 2nd Ed., 1976, McGraw-Hill Book Co.

The syntax for transmitted light is

```
transmission color, scale, ior
```

or

```
transmission scale, ior
```

The *scale* parameter indicates how much light will be transmitted throught the surface. An non-transparent surface uses a value of 0, whereas a fully transparent surface uses 1. The parameter *ior* is the index of refraction (see Table 4.2 for examples).

Microfacet Distribution

As mentioned earlier, the size of the specular highlight is dependent on the roughness of the surface. Figure 4.8 shows the variation in the diffuse and specular reflections as a function of the surface roughness.

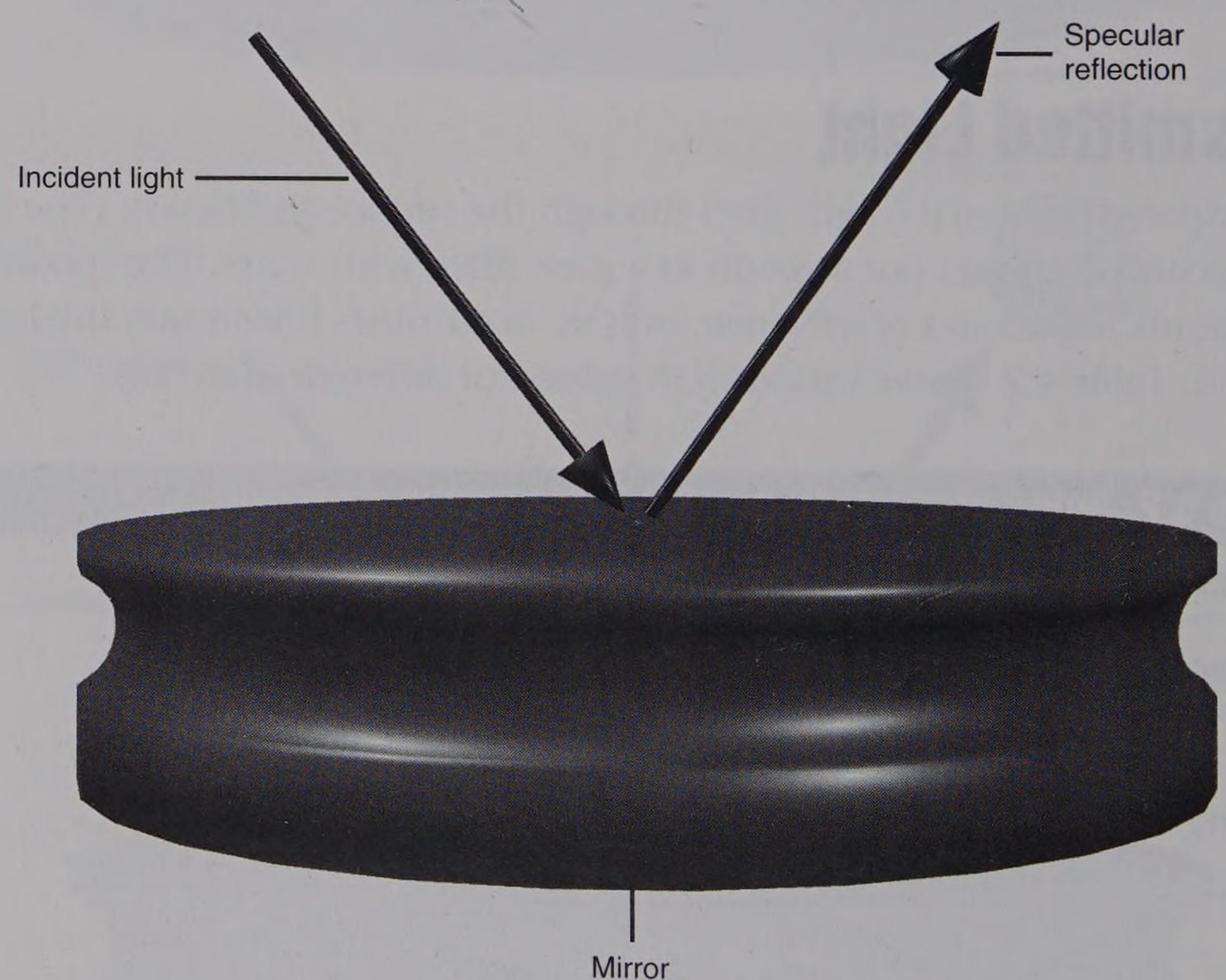
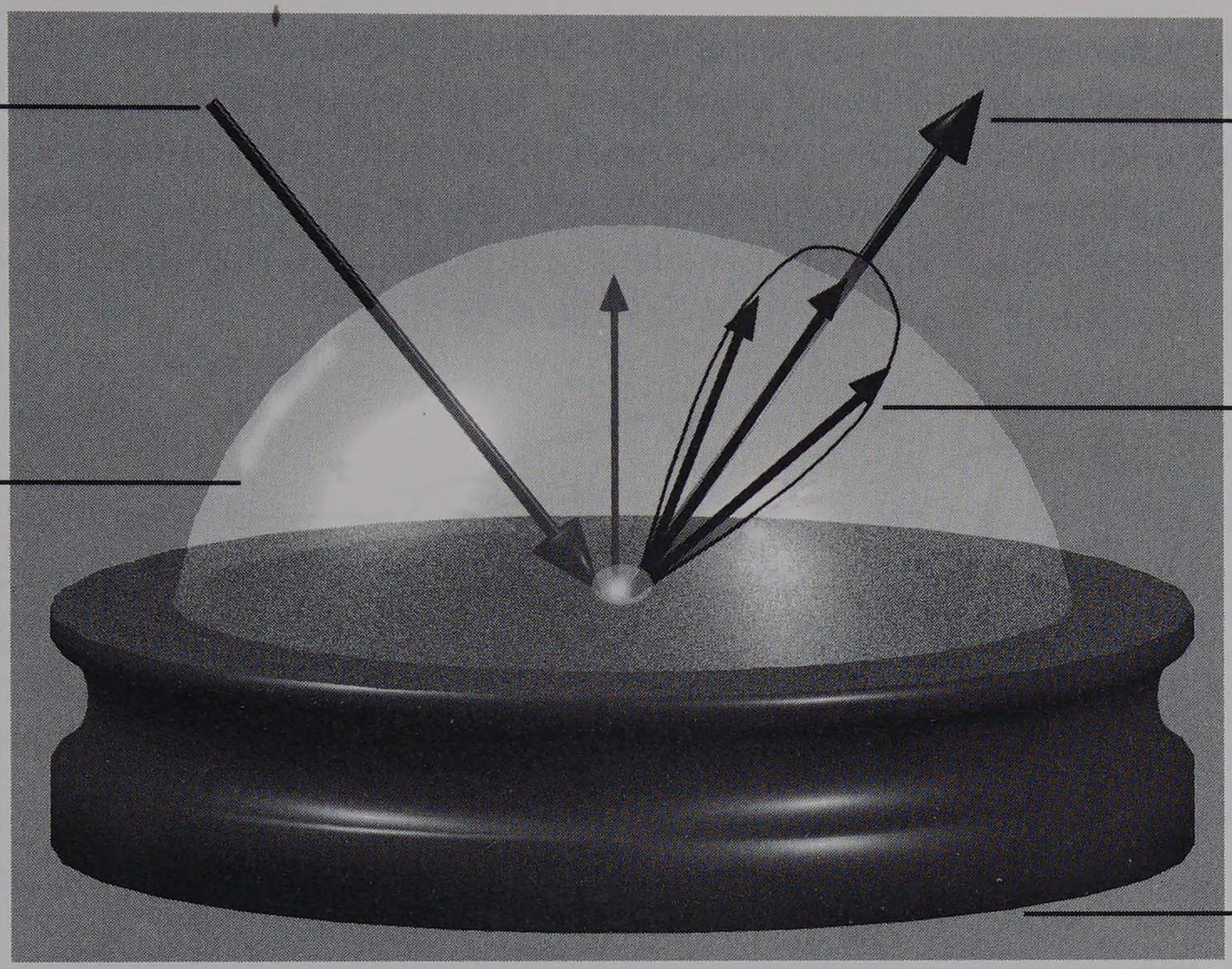
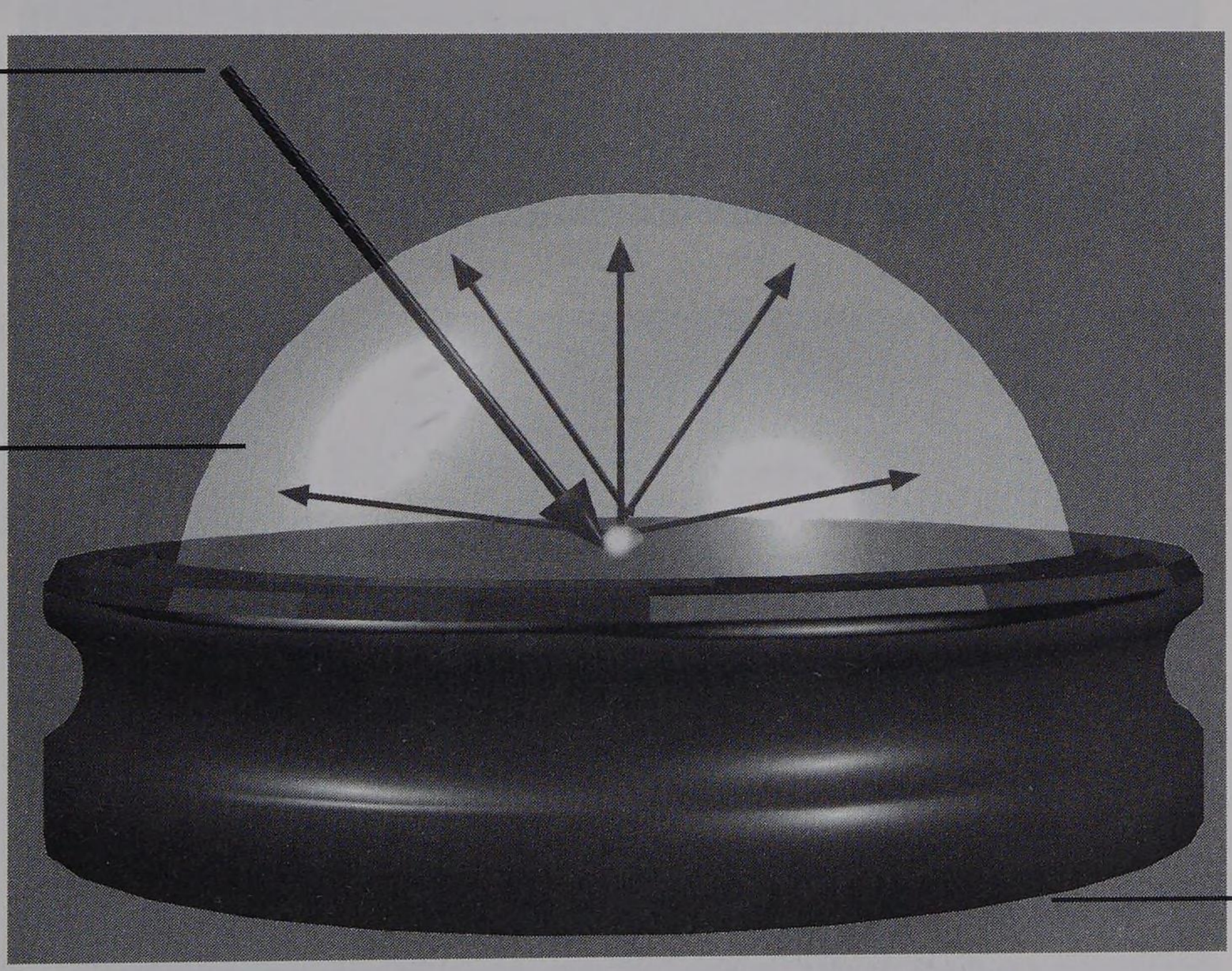


Figure 4.8
Diffuse and specular reflections for different roughness values.

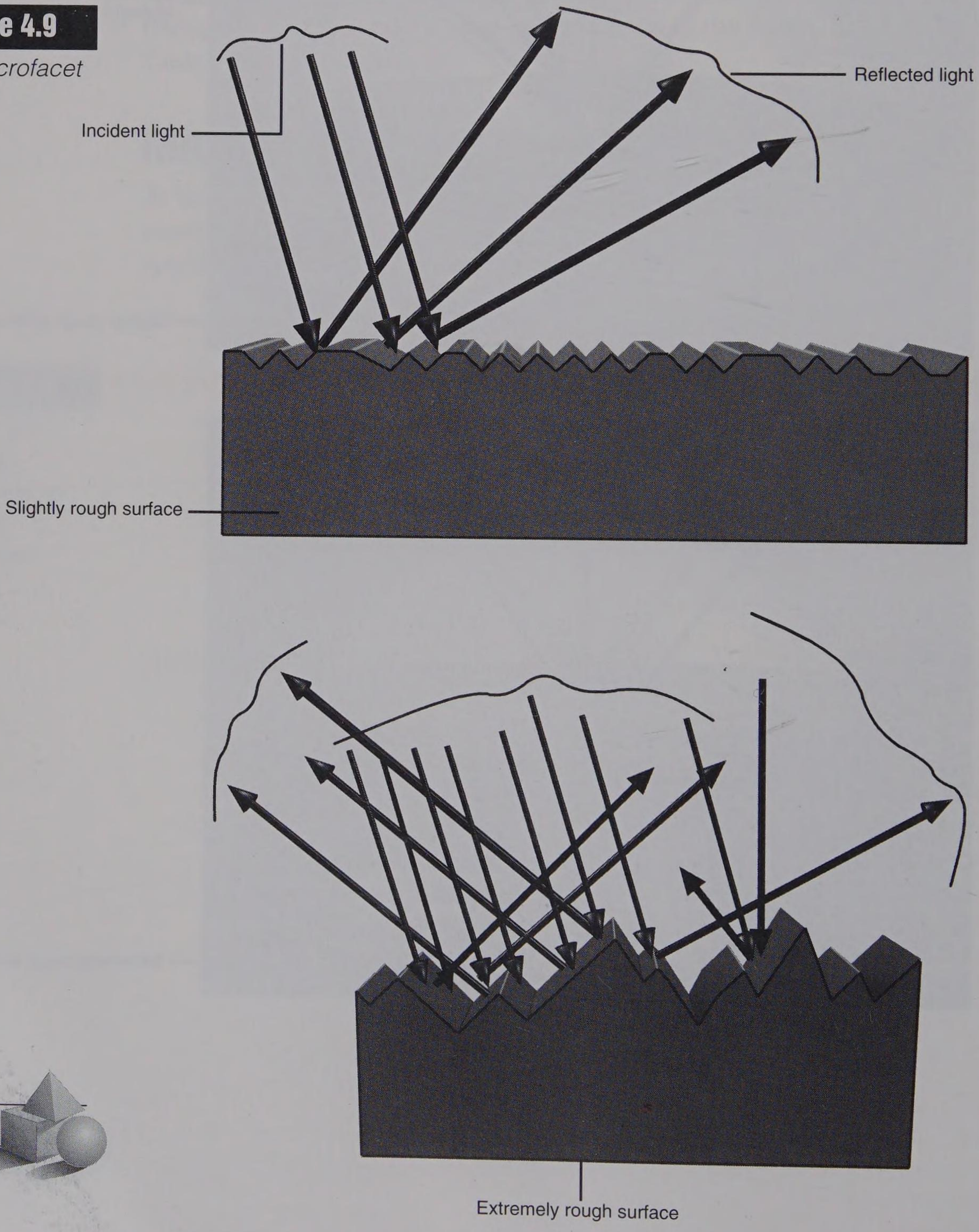

Incident light
Specular reflection
Specular highlight
Diffuse reflection
Slightly rough surface


Incident light
Diffuse reflection
Extremely rough surface

A widely used approach to modeling surface roughness is called the *microfacet model*. In this model, the surface of the object is assumed to consist of very small mirror surfaces that are oriented in all directions. A small roughness value indicates that all the mirrors are basically aligned with the object's surface, whereas a large value indicates that the mirrors are aligned randomly in all directions (see fig. 4.9).

Figure 4.9

The microfacet model.



There are different models available, and each assumes a certain alignment distribution. The choice of a particular model affects the size and shape of the specular highlight. (For more details, read *Advanced Animation and Rendering Techniques*, by A. Watt and M. Watt, 1992, ACM Press, pp. 52-62.)

The syntax Polyray uses to select a microfacet model is

```
microfacet name angle
```

or

```
microfacet angle
```

in which *name* is one of the different models available: Blinn, Cook, Gaussian, Phong (the default), and Reitz. The *angle* parameter is the fall-off angle at which the highlight decreases to 50 percent of its intensity (*angle* must be between 0 and 45 degrees). The greater the value, the rougher the surface.

In the next section, you can compare the different effects that occur when the texture parameters are changed.

Examples

In this section, you can experiment by changing the values of different parameters in the standard shading model. Figure 4.10 is a collection of the images generated by modifying the parameters. In each row, only one parameter is modified.

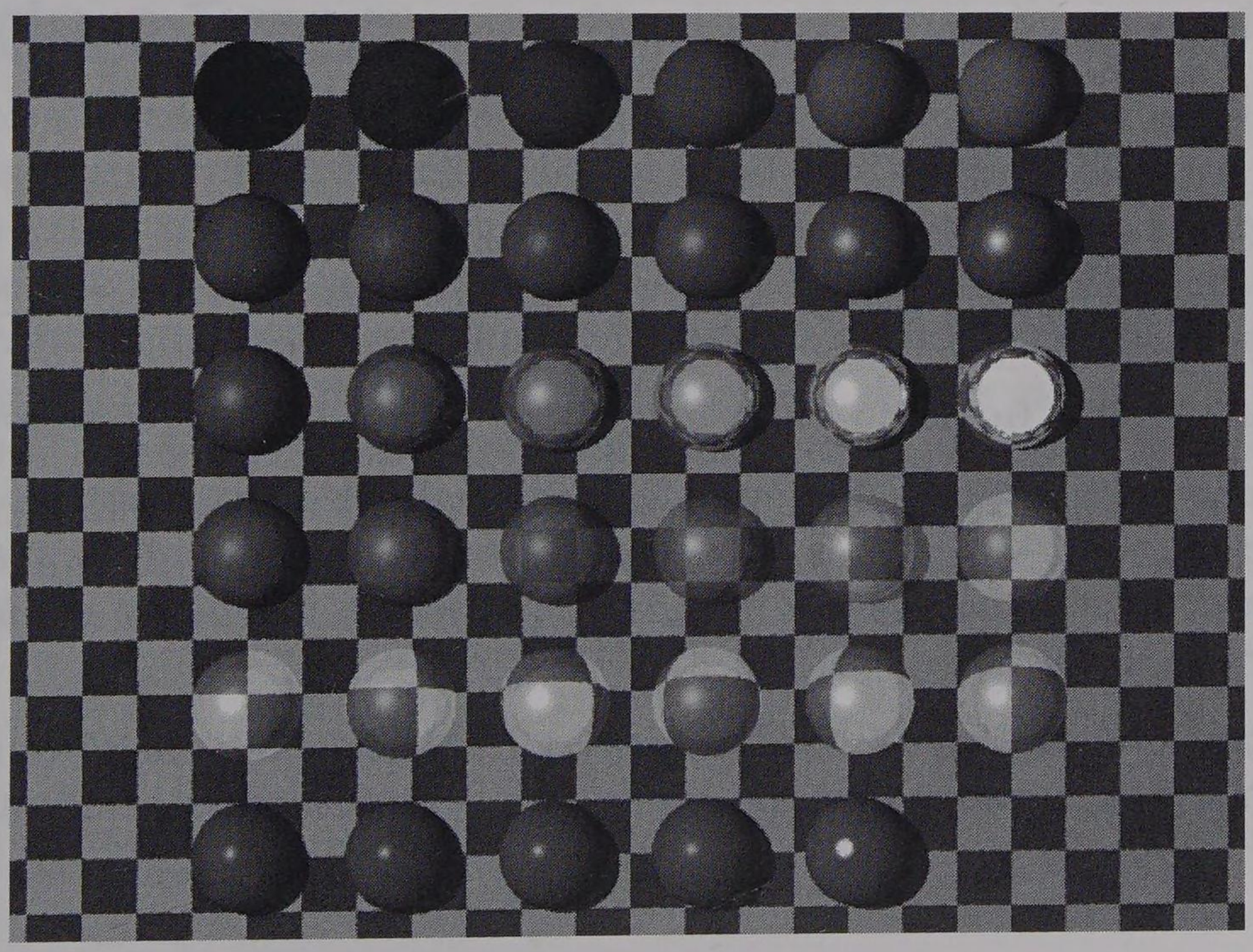


Figure 4.10

Effects of changing parameters in textures.

You can start with a dull, red surface,

```
texture {
      surface {
          color red
          ambient 0.1
          diffuse N
      }
}
```

in which `N` is changed from 0 to 1 in increments of 0.2 (see fig. 4.10, top row).

The sphere changes from almost black to red without highlights (refer to fig. 4.8). The next variation is on the specular highlights. Take the previous definition and add `specular` to it:

```
texture {
      surface {
          color red
          ambient 0.1
          diffuse 0.7
          specular white, N
      }
}
```

in which `N` varies from 0 to 1 in increments of 0.2 (see fig. 4.10, second row).

Because no microfacet model was selected, the default Phong is used. The color white is used to create a white highlight.

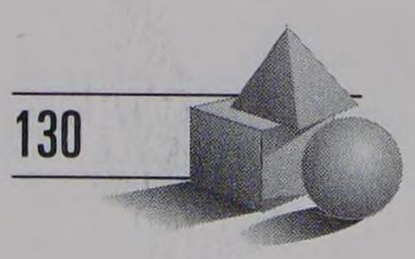
The third row of figure 4.10 shows reflection effects. Again, use the previous definition

```
texture {
      surface {
          color red
          ambient 0.1
          diffuse 0.7
          specular white, 0.5
          reflection white, N
      }
}
```

in which `N` varies from 0 to 1 in increments of 0.2 (see fig. 4.10, third row).

The color white on the reflection indicates that all colors are reflected. Observe how the spheres that are near the last ones on the third row are reflected. Because the sphere is facing the light source, the surface turns into a very light color.

The fourth row of figure 4.10 experiments with the use of the `transmission` parameter. By changing the scale, the spheres become more transparent.



The texture definition used is

```
texture {
      surface {
          color red
          ambient 0.1
          diffuse 0.7
          specular white, 0.5
          reflection white, 0.1
          transmission white, N, 1
      }
}
```

in which `N` varies from 0 to 1 in increments of 0.2 (see fig. 4.10, fourth row).

In this example, the transmission scale is changed and the `ior` remains fixed at 1.0 (`ior` of air). As the objects become more transparent, you can see the undeformed background. If you do the opposite—that is, fix the scale and vary the `ior` value—the result is that of the fifth row. The `ior` value was changed from 1.0 to 3.5 in increments of 0.5. As the `ior` increases, the background is bent more.

Finally, the sixth row in figure 4.10 shows the different microfacet models arranged as Phong, Blinn, Gaussian, Reitz, and Cook. The fall-off angle used was 5 degrees.

This completes a look at the standard shading model. In the next section, you study other types of textures and techniques.

Texture Types

Besides the standard model, Polyray supports procedural, functional, and layered textures. Procedural textures are based on mapping a 2-D pattern to a given surface. A cone with a checkered surface gives the impression that the pattern was projected on the cone's surface. Procedural textures have the mathematical representation of the patterns hard-coded in the program. Functional textures, on the other hand, require that users specify a mathematical expression to evaluate the colors. Therefore, procedural textures are faster to produce than functional textures. Layered textures are used to stack different textures, one on top of the other. You read about these textures later in this chapter.

Checker

The checker texture has been presented in a wide variety of examples throughout this book. It consists of two tiles, each with a different texture. This pattern is repeated throughout the entire surface. The syntax is

```
texture {
      checker texture#1, texture#2
      [transformations]
}
```

in which *texture#1* and *texture#2* are texture declarations or token words representing textures. The *transformations* parameter consists of optional scaling, rotation, and translation operations that are performed on the texture itself. To apply colors to the tiles, you can use the definition presented in the "Colors" section of this chapter.

```
define BlueColor
     texture { surface {color Blue } }

define GreenColor
     texture {surface {color Green} }

texture {
     checker BlueColor, GreenColor
}
```

In some cases, you may want to either rotate the pattern or scale the tiles to create a specific effect, for example, to scale the tiles by half, use the following:

```
texture {
     checker BlueColor, GreenColor
     scale <0.5, 1.0, 0.5>
}
```

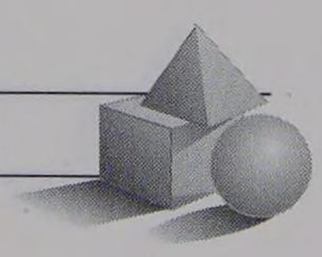
The tiles are scaled by half in the x and z directions.

Hexagon

The hexagon texture is similar to the checker texture in the sense that it repeats a pattern throughout the surface. The tiles are hexagonal and in groups of three. The syntax is

```
texture {
      hexagon texture#1, texture#2, texture#3
}
```

in which *texture#1*, *texture#2*, and *texture#3* are texture declarations or token words representing textures.



Listing 4.6 shows a sphere inside another sphere. The inner sphere is a solid red color and the outer sphere has a hexagonal texture. The texture names for the three tiles used are `Diamond`, `Mirror`, and `Air`. The `Air` texture is really not air, but it's a way of creating an empty space by creating a glass that has an `ior` of 1.0 (`ior` of air) and doesn't bend the images. The floor is composed of hexagons also, with one of the tiles being made out of stone. Figure 4.11 shows the ray-traced image.

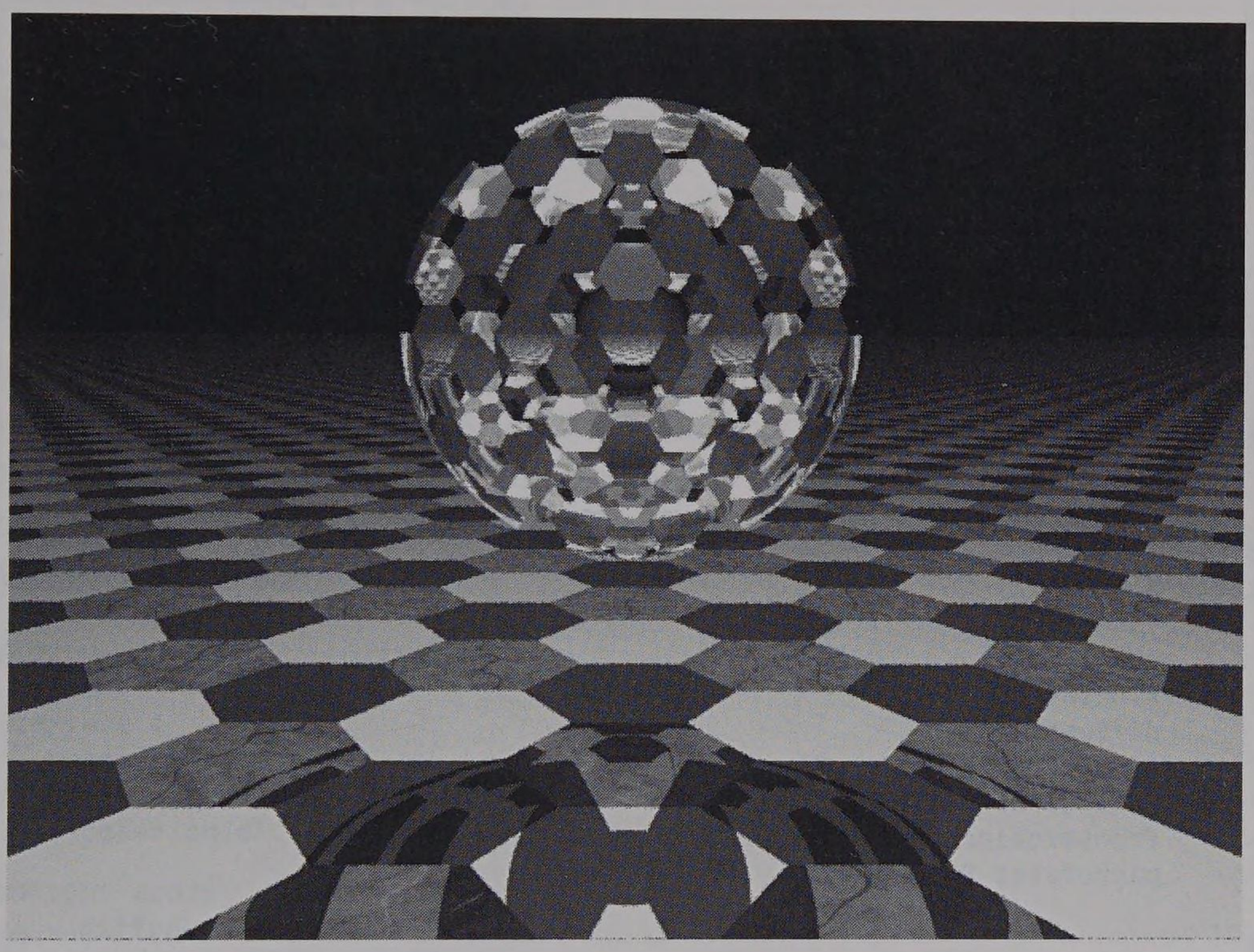
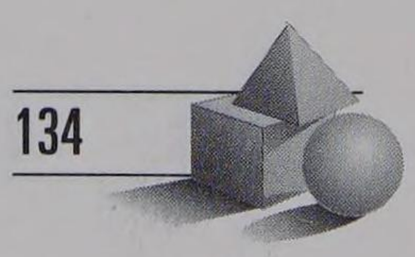


Figure 4.11
Hexagon textures (HEX.PI).

Many things are happening in figure 4.11. First, the `Diamond` and `Air` textures have different `ior` values. The `Diamond` texture bends the light rays and deforms the background. Observe that when you look through the `Diamond` tile, the sphere seems to be smaller than when you look through the `Air` tile. The `Mirror` tile reflects whatever hits the surface at that point. Listing 4.6 presents the scene file used to create figure 4.11.

Listing 4.6. Using the hexagon texture (HEX.PI).

```
// HEX.PI

include "texture.inc"
include "colors.inc"
include "stones.inc"

define Plane_XZ  object {
            polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                       <10000,0,-10000>, <10000,0,10000>
            }

define Air
texture {
   surface {
      color white
      ambient 0
      diffuse 0
      specular 0
      reflection white, 0
      transmission white, 1, 1
     }
}


define Diamond
texture {
   surface {
      color white
      ambient 0.1
      diffuse 0.1
      specular 0.8
      reflection white, 0.1
      transmission white, 1, 2.42
      microfacet Phong 0.5
      }
}


// Set up the camera
viewpoint {
   from <0,0,-10>
   at <0,-1.3,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0
   }

background <0, 0, 0>
light <0,1, -20>
light <0, 10, 0>

object {
   sphere <0, 0, 0>, 2
   texture { hexagon Mirror, Diamond, Air
             rotate <-90, 0, 0>
             scale <0.25, 0.25, 0.25>
```



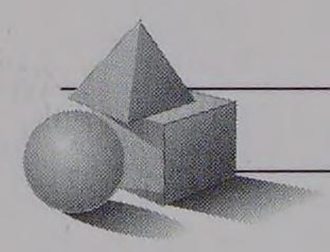
```
   }
}

object {
    sphere <0,0,0>,0.5
    shiny_red
}

object {
   Plane_XZ
   texture { hexagon matte_white, Stone5, matte_black
             scale <1.5,1, 1.5>
   }
   translate <0,-5,0>
}
```

Noise Surface

Noise surface is a general type of texture. You can control the surface of an object in the same manner as the standard shading model. In addition, you can modify the appearance of the surface by introducing noise, bump, dents, and ripples. To increase the randomness of the effects, you can use a turbulence scale value.

The syntax for noise surface is

```
 texture {
   noise surface {
// Standard shading model values
     color color

ambient scale
     diffuse scale
     specular color, scale
     transmission color, scale, ior
     reflection scale
     microfacet name angle
     color_map(map_entries)

// Solid texturing functions
     position_fn index
     position_scale scale
     turbulence scale
     octaves scale
     lookup_fn index

// Surface bump/dent/ripple modifiers
     normal index
     bump_scale scale
     frequency number
     phase scale
   }
 }
```



The first part of the `noise` surface format is identical with that of the standard model, except for the new `color_map` feature. The next sections explain the parameters in detail.

color_map

When a light ray intersects an object, the ray tracer computes an index value. This value is generated by the 3-D texturing function defined by the surface. The number is compared against an internal color lookup table and a color is selected. If you decide to change Polyray's color lookup table (also known as a *color map*), you can do so by creating your own. The syntax for a color map is

```
color_map(
     [start_value, end_value, start_color, end_color]
          ...etc...
)
```

or

```
color_map(
     [start_value, end_value, start_color,start_alpha,
                          ➥end_color, end_alpha]
          ...etc...
)
```

in which *start_value* and *end_value* are numbers that indicate the range of values calculated by Polyray. The values are between 0 and 1. For example:

```
[0, 0.5, Gray, White]
```

If the ray tracer calculates an index value between 0 and 1, it will interpolate between `Gray` and `White` to find the correct color. If the value is 0, use `Gray`. If the value is equal to 1, use `White`.

start_color and *end_color* are related to the start and end values of that segment.

start_alpha and *end_alpha* stand for the amount of transparency in a given color. If alpha is 0, the material is opaque. A value of 1 indicates a transparent color.

For example, Listing 4.7 uses the same noise surface definition for each texture but the color maps are different. The color maps used are for white marbles:

```
color_map(
   [0.0, 0.8, <1, 1, 1>, <0.6, 0.6, 0.6>]
   [0.8, 1.0, <0.6, 0.6, 0.6>, <0.1, 0.1, 0.1>])
```

for red marbles:

```
color_map(
   [0.0, 0.8, <1, 0.8, 0.8>, <1, 0.4, 0.4>]
   [0.8, 1.0, <1, 0.4, 0.4>, <1, 0.1, 0.1>])
```

and for Jade marbles:

```
color_map(
   [0.0, 0.8, <0.1, 0.6, 0.1>, <0, 0.3, 0>]
   [0.8, 1.0, <0.1, 0.5, 0.1>, <0, 0.3, 0>])
```

and for alpha yellow marbles:

```
color_map(
[0.0, 0.2, yellow, 0, yellow, 0]
[0.2, 0.5, yellow, 0, black, 0.2]
[0.6, 1.0, black, 0.2, black, 1])
```

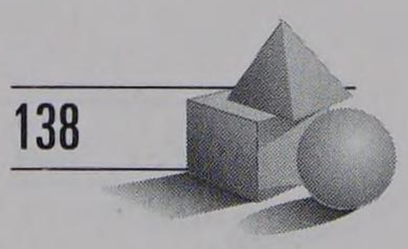
To learn what colors compose a given color map, compare the <R,G,B> values to the colors in Table 4.1. For example, the color <1,1,1> is white, <0.6,0.6,0.6> is a gray tone, near LightGrey, and <0.1,0.1,0.1> is almost black. These three colors help create the white marble texture. If Polyray calculates a value of 0.4, then the color that is halfway between <1,1,1> and <0.6,0.6,0.6> is used.

For the red marble map, observe that the value of red in all the colors is 1 and the rest of the components vary. Thus, most of the tones are based on red, the first of which is near white. For the `Jade` texture, all tones are based on green. Finally, the alpha yellow marble uses alpha values for some of the colors. If the color index is between 0 and 0.2, the surface color is yellow. Between 0.2 and 0.5, the colors vary from yellow (opaque, because alpha is 0) to black (partially transparent because alpha is 0.2). Between 0.6 and 1.0, the color is black and the alpha value goes from 0.2 to 1 (fully transparent). Because a color different than white was used for the transparent areas on the texture, the surface will act as a filter (similar to colored glass). In other words, whatever colors go through the transparent areas will be combined with the color of the surface at that point. To eliminate this feature, the statement `transmission white, 1,1` was added to the texture. The color white indicates that all colors will be able to go through the transparent areas of the texture (no filtering will be performed).

Some of the objects used in Listing 4.7 are Bézier patches. The cone with its apex bent to one of the sides uses the white marble texture. The flat patch uses the red marble texture and the cylinder object uses the Jade. The Bézier patches were created with POVCAD using the Bézier features (see fig. 4.12).

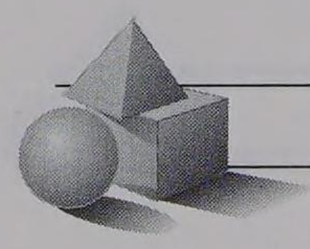
Listing 4.7. Using color maps (COLORMAP.PI).

```
// COLORMAP.PI

include "texture.inc"

define white_marble
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
       [0.0, 0.8, <1, 1, 1>, <0.6, 0.6, 0.6>]
       [0.8, 1.0, <0.6, 0.6, 0.6>, <0.1, 0.1, 0.1>])
      }
}

define red_marble
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
       [0.0, 0.8, <1, 0.8, 0.8>, <1, 0.4, 0.4>]
       [0.8, 1.0, <1, 0.4, 0.4>, <1, 0.1, 0.1>])
      }
}

define Jade
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 5
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
       [0.0, 0.8, <0.1, 0.6, 0.1>, <0, 0.3, 0>]
       [0.8, 1.0, <0.1, 0.5, 0.1>, <0, 0.3, 0>])
      }
```



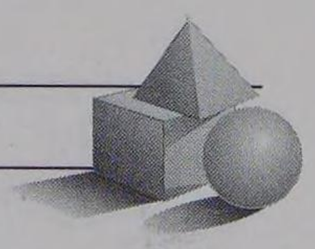
```
}

define alpha_yellow_marble
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      transmission white, 1,1
      color_map(
       [0.0, 0.2, yellow, 0,   yellow, 0]
       [0.2, 0.5, yellow, 0,   black, 0.2]
       [0.6, 1.0, black, 0.2, black, 1]
      )
      }
}

viewpoint {
 from      <-1, 5, -15 >
 at        <2.0, -0.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 200,200
 aspect     1.0
}

// LIGHT_SOURCE
directional_light < 0 ,  5 , -15  >


object{
// BEZIER PATCH CONE
object{
     bezier 2, 0.05, 3, 3,
      <0.183045,2.457585,0.0>, <-0.915,1.755,0.0>,
      <-0.3885,0.702,0.0>, <1.10442,0.0936,-0.013455>,
      <0.183045,2.457585,0.0>, <-1.2075,1.755,0.506025>,
      <-0.94425,0.702,0.962325>, <-0.76875,0.053235,2.644785>,
      <0.183045,2.457585,0.0>, <-1.7925,1.755,0.506025>,
      <-2.05575,0.702,0.962325>, <-2.23125,0.053235,2.644785>,
      <0.183045,2.457585,0.0>, <-2.085,1.755,0.0>,
      <-2.6115,0.702,0.0>, <-3.26319,-1.44261,0.0>
  white_marble
}
+
// BEZIER PATCH CONE
object{
     bezier 2, 0.05, 3, 3,
      <0.183045,2.457585,0.0>, <-0.915,1.755,0.0>,
      <-0.3885,0.702,0.0>, <1.10442,0.0936,-0.013455>,
      <0.183045,2.457585,0.0>, <-1.2075,1.755,-0.506025>,
      <-0.94425,0.702,-0.962325>, <-0.76875,0.013455,-2.63133>,
```

continues



Listing 4.7. Continued.

```
        <0.183045,2.457585,0.0>, <-1.7925,1.755,-0.506025>,
        <-2.05575,0.702,-0.962325>, <-2.23125,0.013455,-2.63133>,
        <0.183045,2.457585,0.0>, <-2.085,1.755,0.0>,
        <-2.6115,0.702,0.0>, <-3.26319,-1.44261,0.0>
  white_marble
}
}

// BEZIER PATCH  FLAT
object{
     bezier 2, 0.05, 3, 3,
      <-3.8536,-3.1144,1.0>, <-2.356,-3.26,1.0>,
      <-0.5296,-3.8632,1.0>, <2.9768,-5.5984,1.0>,
      <-1.0,-2.0,1.7>, <-0.3,-2.0,1.7>,
      <0.3,-2.0,1.7>,  <1.0,-2.0,1.7>,
      <-1.0,-2.0,2.3>, <-0.3,-2.0,2.3>,
      <0.3,-2.0,2.3>, <1.0,-2.0,2.3>,
      <-1.9728,-0.0088,3.0>, <0.9496,0.776,3.0>,
      <2.7576,1.324,3.0>, <3.7808,0.2832,3.0>
  scale<2,1,2>
  translate<2.5, -1.5, 0>
  red_marble

}

object{
// BEZIER PATCH  CYLINDER
object{
     bezier 2, 0.05, 3, 3,
      <2.0,0.0,0.0>, <-0.0864,-2.8496,-1.0>,
      <2.9816,-3.0688,-1.0>, <6.5432,-2.5568,0.0>,
      <2.0,0.5,0.0>, <2.0,0.5,-1.0>,
      <4.0,0.5,-1.0>, <4.0,0.5,0.0>,
      <2.0,1.0,0.0>, <2.0,1.0,-1.0>,
      <4.0,1.0,-1.0>, <4.0,1.0,0.0>,
      <1.3016,2.1008,0.0>, <-0.6344,0.7488,-1.0>,
      <4.0,1.5,-1.0>, <5.612,1.936,0.0>
  Jade
}
+
// BEZIER PATCH  CYLINDER
object{
     bezier 2, 0.05, 3, 3,
      <6.5432,-2.5568,0.0>, <4.0,0.0,1.0>,
      <2.0,0.0,1.0>, <2.0,0.0,0.0>,
      <4.0,0.5,0.0>, <4.0,0.5,1.0>,
      <2.0,0.5,1.0>, <2.0,0.5,0.0>,
      <4.0,1.0,0.0>, <4.0,1.0,1.0>,
      <2.0,1.0,1.0>, <2.0,1.0,0.0>,
      <5.612,1.936,0.0>, <6.3056,3.324,1.0>,
      <2.324,3.5616,1.0>, <1.3016,2.1008,0.0>
  Jade
```



```
}
+
object {
   polygon 4, <-1,1,0>, <1,1,0>, <1,-1,0>, <-1,-1,0>
   scale <2, 2, 2>
   translate<0,0,-8>
   alpha_yellow_marble
}
 translate<2, 0, 0>
}
```

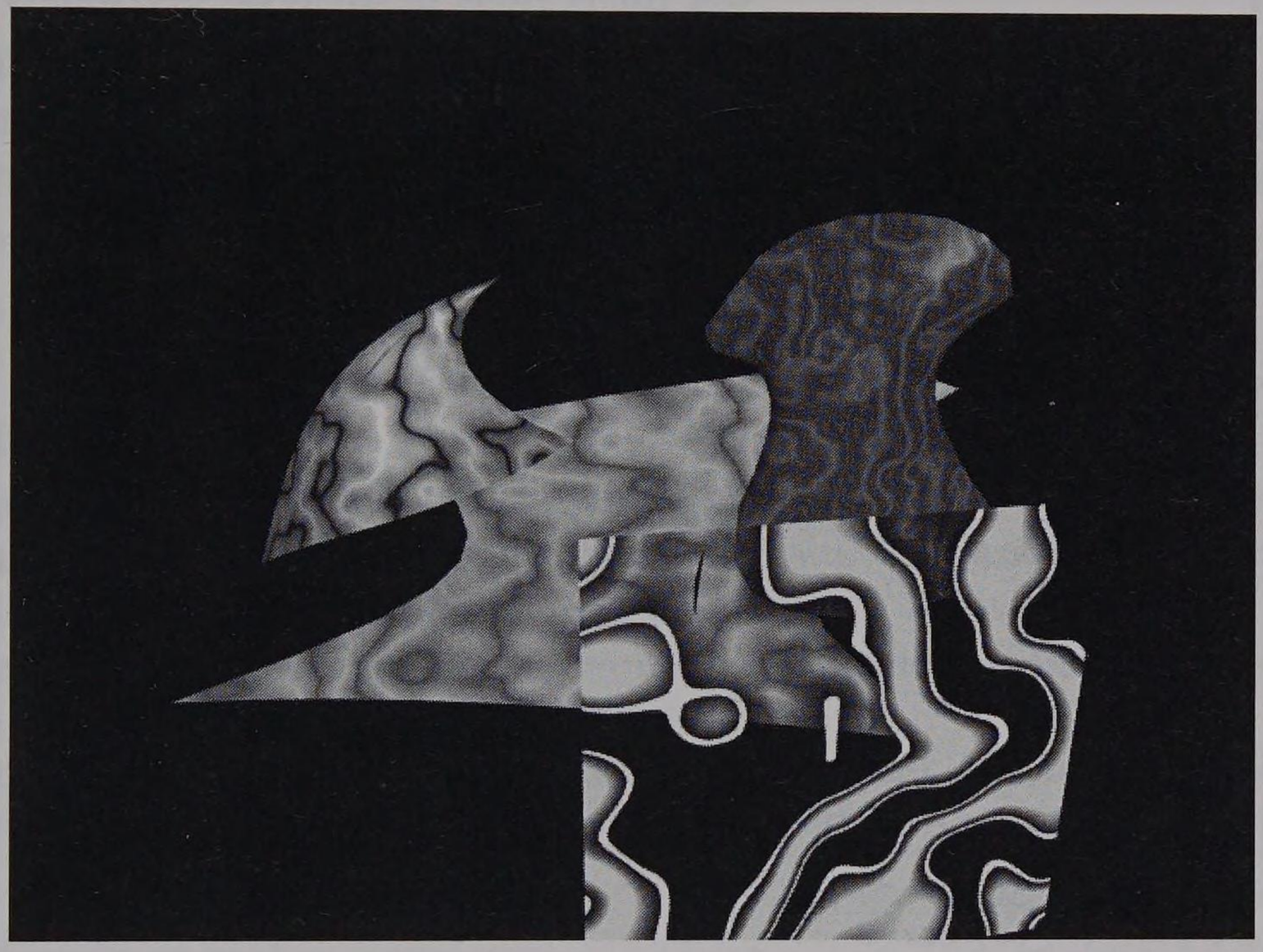


Figure 4.12
Applying the same textures with different color maps—white marble, red marble, jade, and alpha yellow marble (COLORMAP.PI).

There is an excellent utility called CMAPPER that composes different color maps visually (check the Introduction for bulletin boards that carry CMAPPER).

The next sections present other parameters used by the Noise Surface texture definition—solid texturing functions. They are used to modify and add noise to the texture patterns.

position_fn

When a light ray intersects an object, Polyray computes the location of that intersection point. When the intersection point is defined in terms of a coordinate system that is attached to the object, it is called a *local* coordinate system. If the point is defined in terms of the coordinate system that

you normally use to locate geometry, it is called the *global* system. You can use the location of the intersection point to modify the color. The keyword `position_fn` lets you select how to use that point to affect the color calculation. The available values for `position_fn` are

Index	Method
0	Default. None.
1	Use the X value in the local coordinate system (object x).
2	Use the X value in the global coordinate system (world x).
3	Use the distance from the Z axis (cylindrical).
4	Use the distance from the origin (spherical).
5	Use the distance of the point around the Y axis. The coordinate system goes from 0 to 1 counterclockwise (radial).

The file TEXTURE.INC defines the following token names, which can be used in place of the index values:

```
define position_plain       0
define position_objectx     1
define position_worldx      2
define position_cylindrical 3
define position_spherical   4
define position_radial      5
```

Examples of this function are given later.

position_scale

If you select a parameter for `position_fn` other than 0, you can use a scale factor to increase or decrease the effect. The keyword `position_scale` affects the contribution of `position_fn`. As a result, the contribution is equal to `position_fn * position_scale`.

turbulence

Turbulence is the amount of distortion that can be introduced into the calculations. The distortion is produced by a noise function. The `turbulence` value can be looked at as a scaling factor for the noise function; in other words,

```
noise contribution = turbulence * noise3D
```

in which `noise3D` is Polyray's internal noise function generator.

octaves

`octaves` is the number of times that the `noise` function is called. By increasing the number of octaves, the amount of detail in the texture increases. The `noise3D` function is dependent on the `octaves` value.

```
noise contribution = turbulence * noise3D(octaves)
```

lookup_fn

The final value of the color depends on the contribution of `position_fn`, `position_scale`, `turbulence`, and `octaves`. The value that results from these is modified in turn by a procedure that uses different kinds of functions: `sawtooth`, `sine`, and `cosine`. The keyword to define the function used is `lookup_fn`, in which the values are as follows:

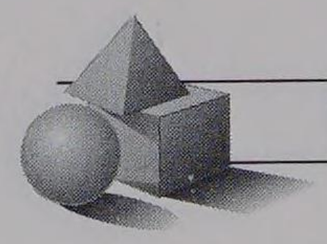
Index	Function
0	No modification (plain)
1	`sawtooth`
2	`cosine`
3	`sine`

The file TEXTURE.INC has defined token words that can be used in place of the lookup functions:

```
define lookup_plain    0
define lookup_sawtooth 1
define lookup_cos      2
define lookup_sin      3
```

All the lookup functions return a number between 0 and 1.

Figure 4.13 shows an array of spheres with a white marble texture. For each row, the `position_fn` value is constant and the `lookup_fn` function is changed.



The arrangement is

```
Row (position_fn)   Column(lookup_fn)
                  plain     sawtooth  cos  sin
plain
objectx
worldx
cylindrical
spherical
radial
```

Observe that with `lookup_fn 0` the spheres have a color (the first color in the map) and no pattern.

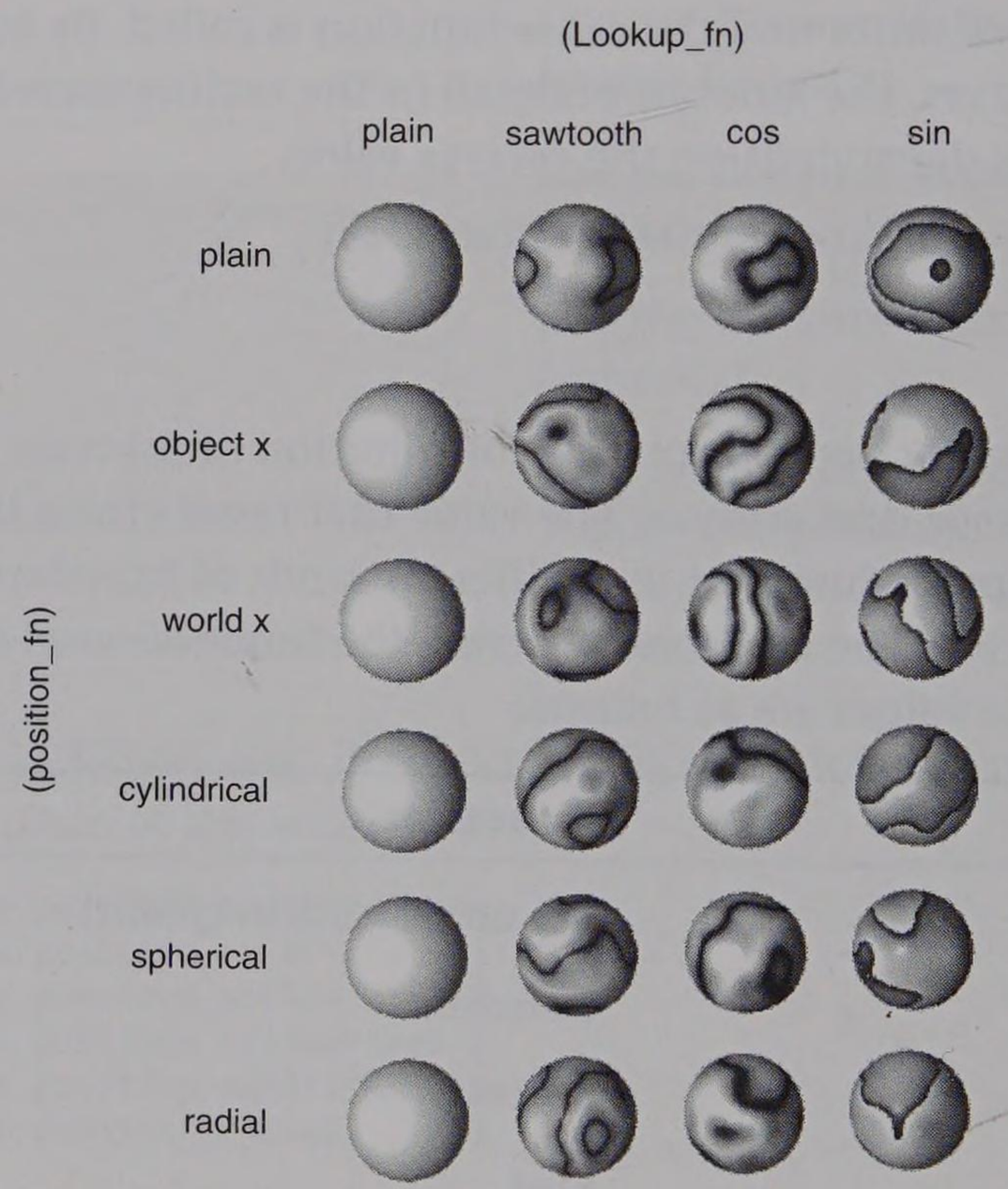


Figure 4.13
Variation in position and lookup functions.

Figure 4.14 shows the same image, this time showing the relationship between `turbulence` and `octaves`. `turbulence` varies from 1 to 4 in increments of 1 (left to right) and the `octaves` vary from 1 to 11 in increments of 2 (top to bottom). For this figure, `position_fn` = 1, `lookup_fn` = 1 and `position_scale` = 2.

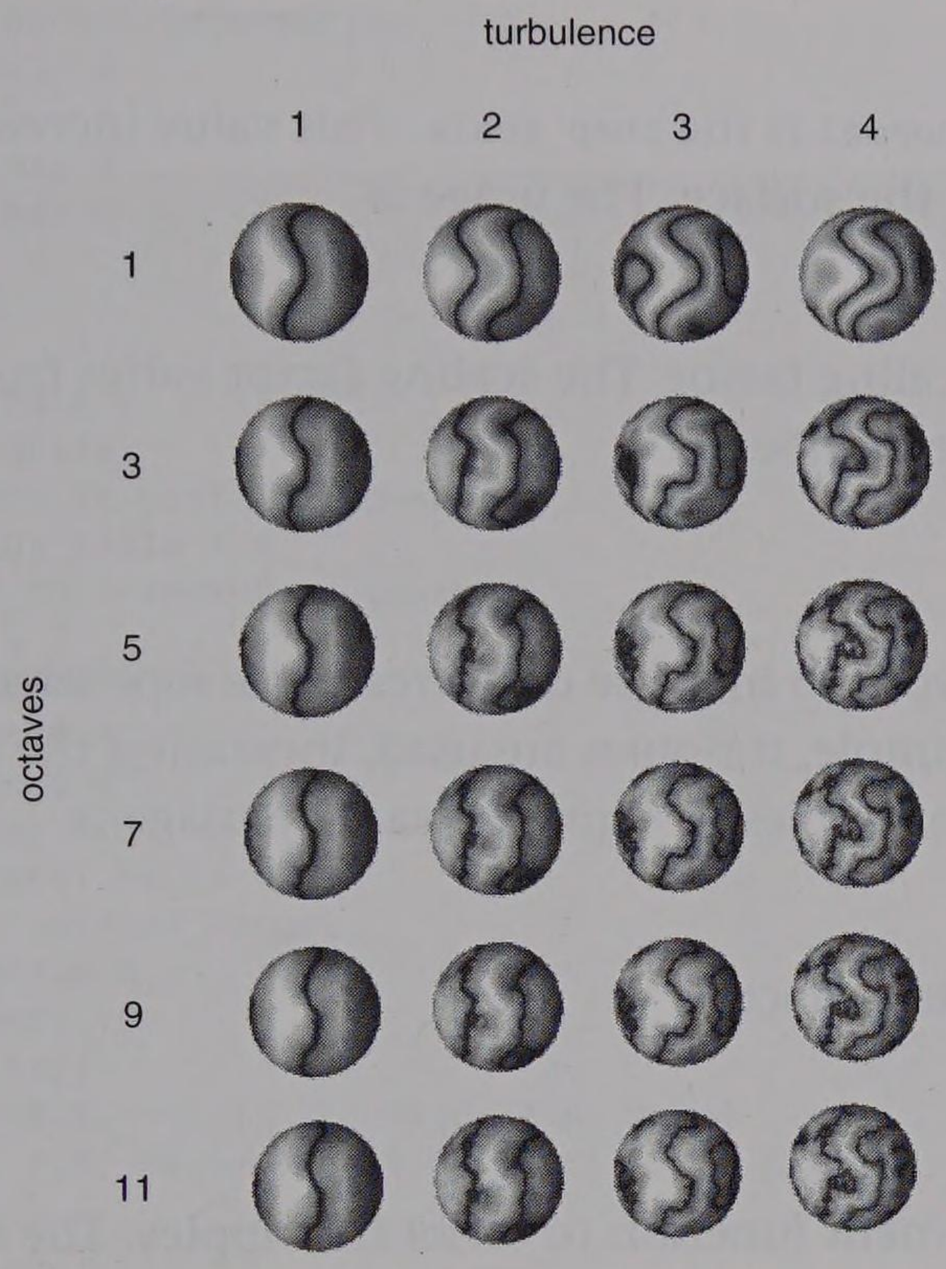


Figure 4.14

The relationship between octaves and turbulence—turbulence varies from left to right; octaves vary from top to bottom.

normal

The keyword `normal` is used to specify if a virtual surface deformation is done on the object. The deformation is called virtual because the surface will not be deformed, but the calculations the ray tracer does will assume that it is. There are three types of deformation methods: `bumps`, `ripples`, and `dents`. The usage for `normal` is

```
normal type
```

in which *type* stands for the type of deformation. Possible values for *type* are

Value	Type
0	Plain (no deformation)
1	bumps
2	ripples
3	dents

bump_scale

The scaling factor for `normal` is the `bump_scale`. This value increases the effect of the deformation on the surface. The usage is

```
bump_scale scale
```

in which *scale* is the scaling factor. The scaling factor varies from 0 (no bumps) to any positive number.

frequency

Use the `frequency` keyword to increase or decrease the repetitiveness of ripple patterns. For example, if ripples are used, increasing the `frequency` increases the number of ripples in a given area. The usage is

```
frequency n
```

in which *n* is the repetition factor.

phase

`phase` acts as a displacement function to offset the ripples. The syntax is

```
phase n
```

in which *n* is the amount of displacement.

Surface Normal Modifier Examples

Following are a few examples of the use of surface normal modifiers. Listing 4.8 shows an attempt to create the moon (see fig. 4.15). If you look closely, the `noise surface` declaration is similar to that of the white marble texture. The `turbulence` value was lowered to produce a subtle effect. To deform the surface lightly, I used the following parameters:

```
normal dented_normal
bump_scale 0.2
frequency 10
```

I used these parameters in an attempt to simulate the deformed surface of the moon. Feel free to experiment with the texture and improve the look.

Listing 4.8. Moon with surface deformation and turbulence (MOON.PI).

```
// MOON.PI

include "texture.inc"
include "colors.inc"
```

```
// Types of Normal Deformation
//plain_normal  0
//bump_normal   1
//ripple_normal 2
//dented_normal 3

define Moon
texture {
   noise surface {
      color white
      position_fn position_spherical
      position_scale 1.0
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 0.53
      ambient 0.4
      diffuse 0.8
      specular 0.2
      microfacet Reitz 5
      normal  dented_normal
      bump_scale 0.2
      frequency 10
      color_map(
       [0.0, 0.8, <1, 1, 1>, <0.6, 0.6, 0.6>]
       [0.8, 1.0, <0.6, 0.6, 0.6>, <0.1, 0.1, 0.1>])
      }
}


// Set up the camera
viewpoint {
   from <0,0,-15>
   at <0.5,0,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0
   }

background <0, 0, 0>
light <-10, 3, -20>

object {
   sphere <0, 0, 0>, 5
   Moon
}
```

The next example uses ripples to create a water-like surface. Three spheres are also in the scene, and you can see their reflections in the water (see Listing 4.9 and the output in fig. 4.16).

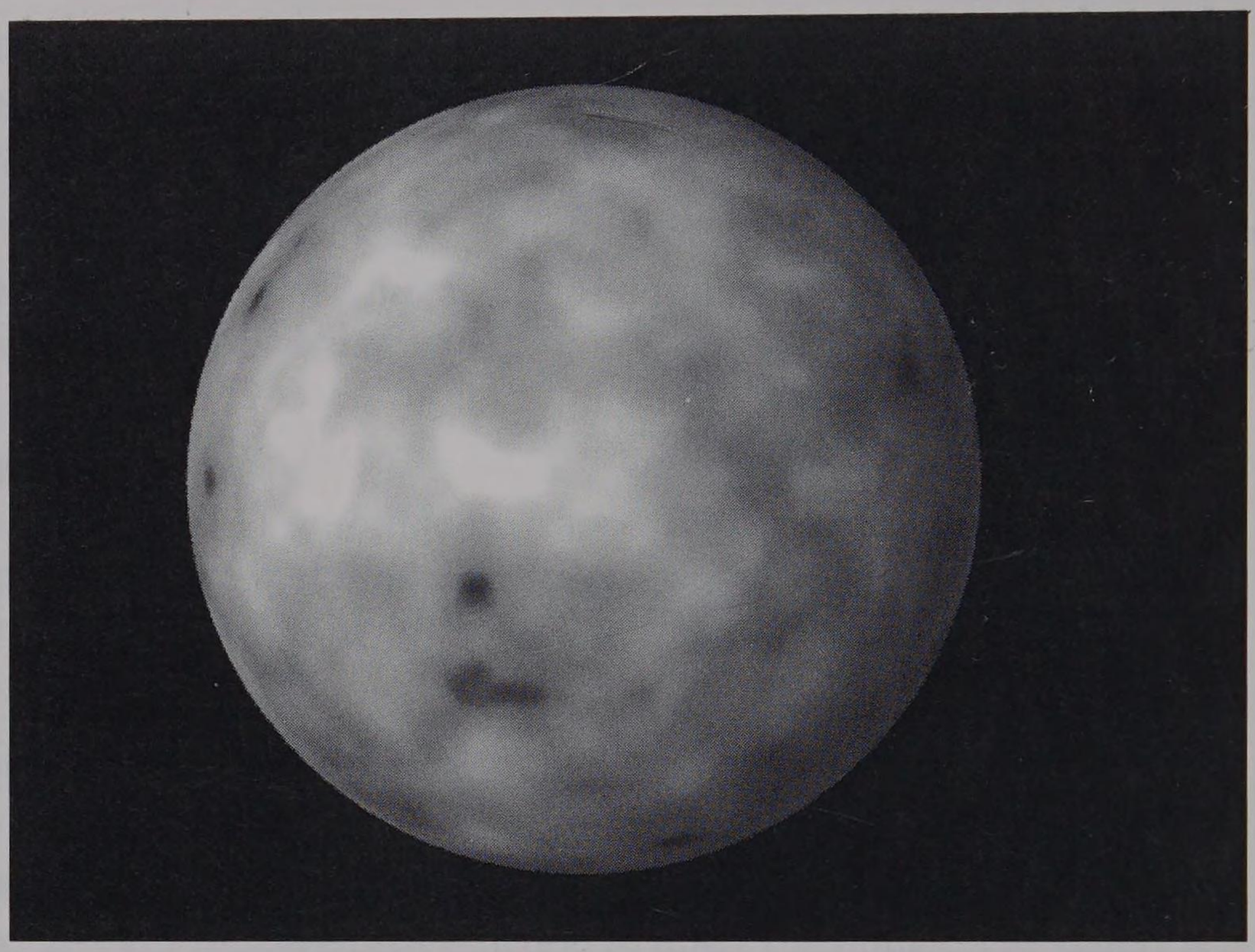


Figure 4.15

Using a lightly dented surface to model the moon (MOON.PI).

Listing 4.9. Modeling an ocean.

```
// WATER.PI

include "texture.inc"
include "colors.inc"

define Plane_XZ  object {
            polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                       <10000,0,-10000>, <10000,0,10000>
          }

define Water
texture {
   noise surface {
      color <0.4, 0.4, 1.0>
      normal ripple_normal
      frequency 100
      bump_scale 1
      ambient 0.3
      diffuse 0.3
      specular white, 0.7
      reflection 0.9
      microfacet Reitz 10
      }
      scale<10, 1, 10>
   }
```

```
// Set up the camera
viewpoint {
   from <0,5,-15>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0
   }
background White
directional_light <0, 3, -10>

object {
   sphere <0, 1.5, 0>, 2
   shiny_red
}
object {
   sphere <-4.5, 3, 1>, 1
   shiny_green
}
object {
   sphere <4.5, 4, 2>, 1
   shiny_yellow
}


object {
   Plane_XZ
   Water
}
```

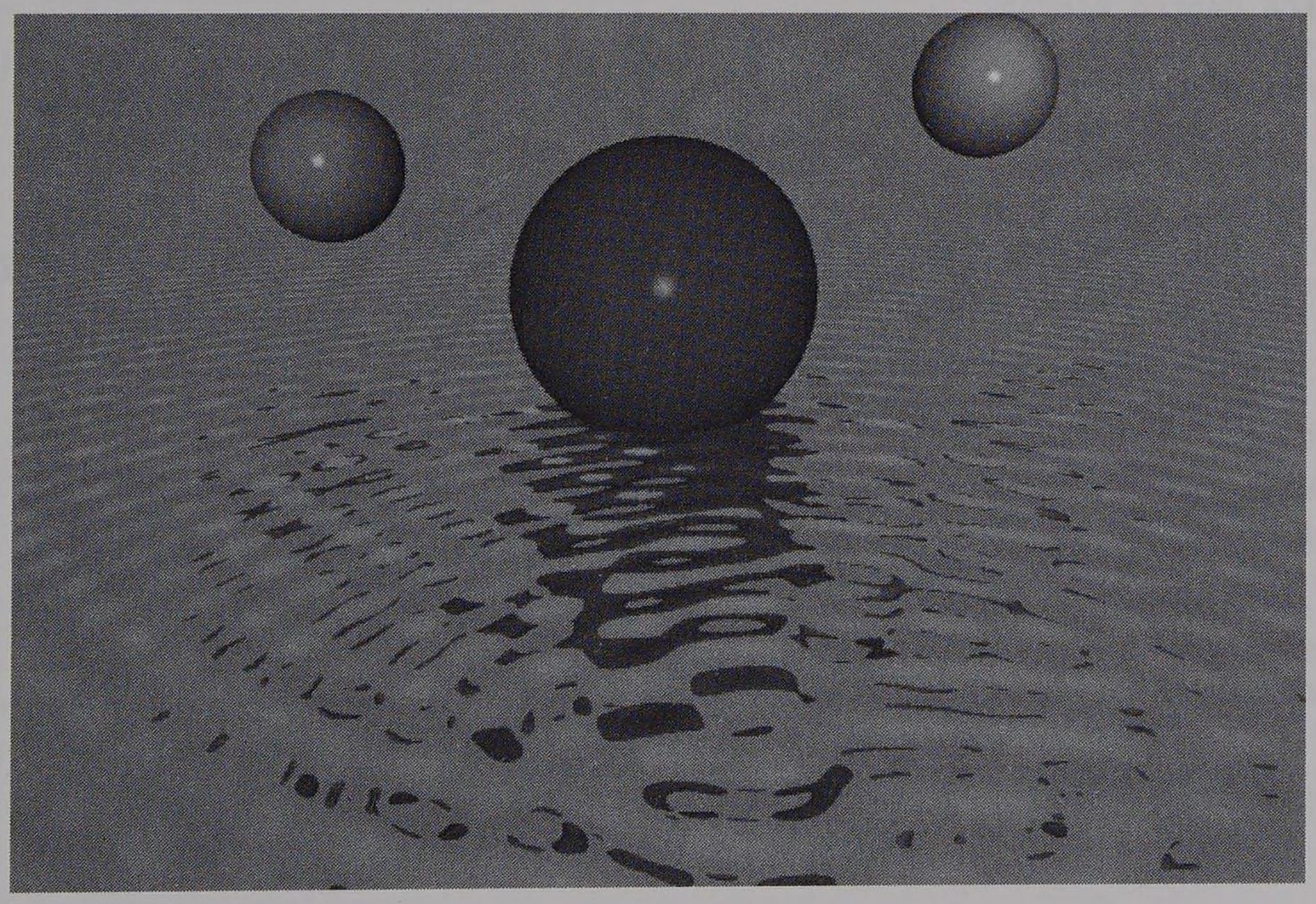


Figure 4.16

Modeling ripples in water (WATER.PI).

The statements used in the scene to create the ripples are

```
normal ripple_normal
frequency 100
bump_scale 1
```

A `scale <10,1,10>` statement is used to separate the ripples in the surface. Another interesting effect is to add transparency to the water and define an `ior`. To create the effect of depth, you can put a second plane with a darker blue tone under the water plane.

The final example uses dents. In Listing 4.10, two of the objects have dented textures. The back plane is a dented mirror, which shows irregular reflections. The red sphere in the middle of the scene is also dented, and it is used to show contrast with the other two spheres, which are smooth (see fig. 4.17).

Listing 4.10. Mirror and red sphere with dented surface (DENTS.PI).

```
// DENTS.PI

include "texture.inc"
include "colors.inc"

define Plane_XZ  object {
            polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                       <10000,0,-10000>, <10000,0,10000>
          }

define dented_mirror
texture {
   noise surface {
      normal dented_normal
      frequency 10
      bump_scale 2
      ambient white, 0.1
      diffuse white, 0.2
      specular 0
      reflection  white, 1
      microfacet Reitz 10
      }
   scale <5, 5, 1>
   }

define dents
texture {
   noise surface {
      color <1, 0, 0>
      normal dented_normal
```

```
      frequency 2
      bump_scale 3
      ambient white, 0.2
      diffuse white, 0.5
      specular 0.7
      microfacet Reitz 10
      }
   scale <0.2, 0.2, 0.2>
   }

// Set up the camera
viewpoint {
   from <0,5,-15>
   at <0,2,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0

   }

background <1, 1, 1>
light <0, 3, -10>
light <0, 3, 4>

object {
   sphere <0,0,0>, 1.5
   dents
}
object {
   sphere <-3,0,0>, 1.5
   shiny_green
}
object {
   sphere <3,0,0>, 1.5
   shiny_yellow
}

object {
   Plane_XZ
   texture { checker matte_white, matte_black }
}

object {
   Plane_XZ
   rotate <90, 0, 0>
   translate<0,0, 8>
   dented_mirror
}
```

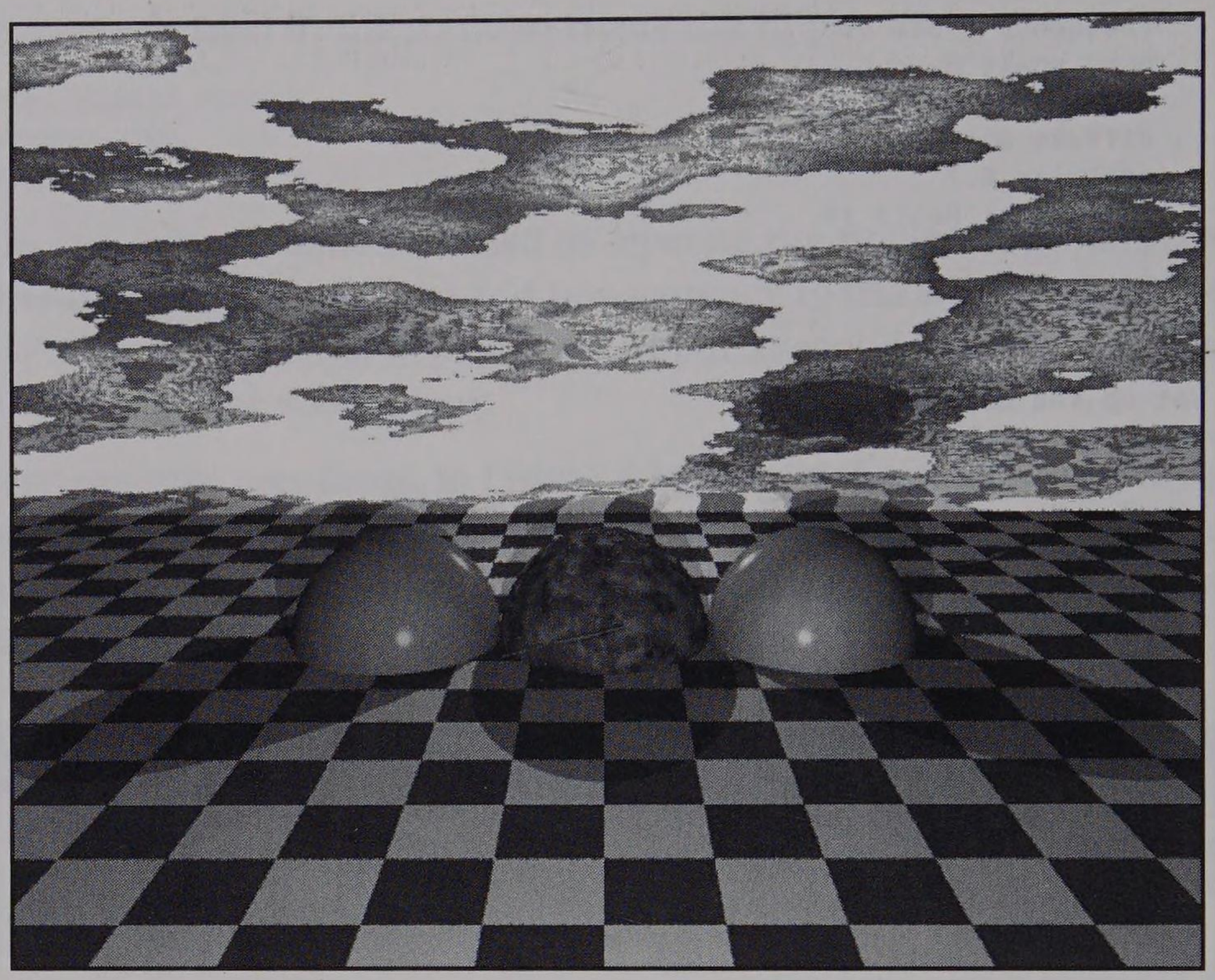


Figure 4.17
A dented mirror and sphere (DENTS.PI).

Layered

Polyray can stack different types of textures on top of each other. The resulting texture is called a *layered* texture. The syntax for constructing layered textures is

```
texture {
      layered
          texture#1,
          texture#2,
          .....etc.....
          texture#N
}
```

in which `texture#1` is the top layer and `texture#N` is the bottom layer.

For layered textures to work correctly, the textures must be transparent either through transmission or by including alpha values in the color maps. The idea is to make the top layers transparent and the final layer opaque. The result is a combination of the textures to make the final color. Listing 4.11 presents an example that is a combination of transparent yellow marble and checker textures. The transparent yellow marble texture is located on the top layer, and the checker texture is last. The checker pattern will be visible through the transparent areas on the top layer. Figure 4.18 presents a plane and a sphere with the created texture.

Listing 4.11. Layered texture with transparent yellow marble and checkers.

```
// LAYERED.PI

include "texture.inc"
include "colors.inc"

background white

define alpha_yellow_marble
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      transmission white, 1,1
      color_map(
       [0.0, 0.2, yellow, 0,   yellow, 0]
       [0.2, 0.5, yellow, 0,   black, 0.2]
       [0.6, 1.0, black, 0.2, black, 1]
      )
      }
}

define Yellow_Marble_Checker
   texture {
       layered
       alpha_yellow_marble,
       texture {checker matte_white, matte_black
             scale <0.2, 0.2, 0.2>
       }
    }

viewpoint {
 from      <-2, 3, -7 >
 at        <0.0, -0.2, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}

// LIGHT_SOURCE
directional_light < 0 ,  5 , -7  >


object {
   polygon 4, <-1,1,1>, <1,1,1>, <1,-1,1>, <-1,-1,1>
   scale <2, 2, 2>
   Yellow_Marble_Checker
}
object {
   sphere <0,0,-3>, 1
   Yellow_Marble_Checker
}
```

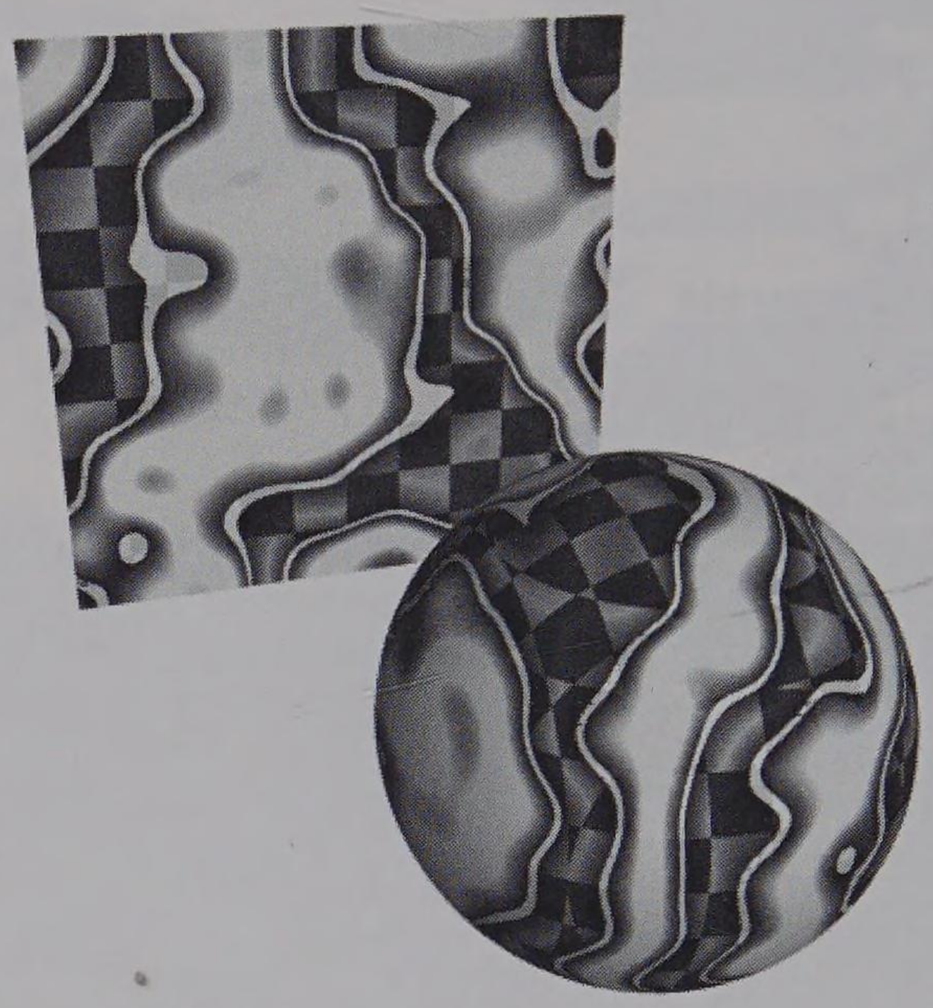


Figure 4.18
Using layered textures (LAYERED.PI).

Functional

Functional textures are evaluated at run time; they are not hard-coded in the ray tracer program. You can use a mathematical expression to determine the color or to deflect the normal of the surface (Chapter 6, "Additional Features," goes into more detail about mathematical expressions). You shouldn't be concerned with the mathematics of color definition or normal deflection because Polyray has a set of defined functions ready for you to use. The lookup functions, which are listed next, are really token names for mathematical expressions. The expressions are given in the COLORS.INC file. The following are the defined token names for the functions:

```
agate_fn        marble_fn         granite_fn      wood_fn
odd_wood1       noisy_wood_fn     bozo_fn         leopard_fn
onion_fn        noisy_onion_fn    checker_fn      checker3_fn
radial_fn
```

After reading Chapter 6, you will be able to create your own functions.

The syntax for a functional texture is

```
texture {
     special surface {
          parameter declaration
     }
}
```

in which the *parameter declaration* is similar to the noise surface.

The primary difference between noise surface texture and functional texture is that you can use any function to look up the colors in the color map. On the other hand, the noise surface texture has a limited set of lookup functions, and those are built into the program.

Listing 4.12 uses two color lookup functions: `bozo_fn` and `granite_fn` (see fig. 4.19 for the output). These functions return a value between 0 and 1. In order to find the correct color, a color map must be defined. The lookup function and the color map are used as follows

```
define BlueSkyMap
     color_map(
       [0.0, 0.5, <0.25,0.25,0.5>, <0.25,0.25,0.5>]
       [0.5 , 0.6, <0.25,0.25,0.5>, <0.7,0.7,0.7>]
       [0.6,  1.0, <0.7,0.7,0.7>, <0.3,0.3,0.3>])

define Sky
texture {
   special surface {
      color BlueSkyMap[bozo_fn]
      .....etc.....
      }
    .....etc.....
   }
```

As you can see, the color map is defined like any other color map. When we get to the `Sky texture` definition, the statement `color BlueSkyMap[bozo_fn]` declares that the lookup function `bozo_fn` is used to generate an index value that will be used to select a color from the `BlueSkyMap` color map. The same method is used with the `PinkGranite` definition.

Initially, the `bozo_fn` function returns a number between 0 and 1. That value is then compared against the `BlueSkyMap` color map and a color is chosen. Finally, the result of that operation is given to the keyword `color` as an <R,G,B> vector.

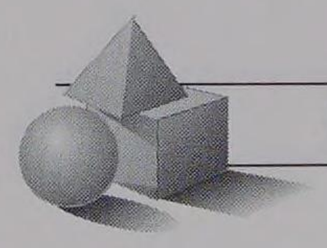
Another interesting method is to declare the `color` as

```
color <expression#1, expression#2, expression#3>
```

in which *`expression#1`*, *`expression#2`*, and *`expression#3`* are mathematical expressions. Each is responsible for calculating one of the color's components. For example, the expression

```
color <agate_fn , marble_fn, granite_fn>
```

indicates that the contribution of the color red will be given by `agate_fn`, the green component by `marble_fn`, and the blue component by `granite_fn`.



Listing 4.12. Using functional textures (SPECIAL.PI).

```
// SPECIAL.PI  Functional Textures

include "texture.inc"
include "colors.inc"

define Quartz   <0.85, 0.85, 0.95>
define Feldspar <0.82, 0.57, 0.46>

define Plane_XZ  object {
           polygon 4, <-10000,0,10000>, <-10000,0,-10000>,
                      <10000,0,-10000>, <10000,0,10000>
         }

define PinkGraniteMap
      color_map(
        [0,    0.4 , Black ,   Black]
        [0.4,  0.45, Quartz,   Feldspar]
        [0.45, 0.5 , Quartz,   Gray]
        [0.5 , 0.55, Gray,     Feldspar]
        [0.55, 0.8,  Feldspar, Feldspar]
        [0.8,  1.0,  Feldspar, Orange])

define BlueSkyMap
      color_map(
        [0.0, 0.5, <0.25,0.25,0.5>, <0.25,0.25,0.5>]
        [0.5 , 0.6, <0.25,0.25,0.5>, <0.7,0.7,0.7>]
        [0.6,  1.0, <0.7,0.7,0.7>, <0.3,0.3,0.3>])


define Granite
texture {
   special surface {
      color PinkGraniteMap[granite_fn]
      ambient 0.3
      diffuse 0.8
      specular 0.3
      reflection  white, 0.3
      microfacet Reitz 10
      }
   }

define Sky
texture {
   special surface {
      color BlueSkyMap[bozo_fn]
      ambient 0.3
      diffuse 0.8
      specular 0.3
```

```
      turbulence 100
      octaves 3
      microfacet Reitz 10
      }
      scale <15, 1, 1>
      translate<0, 0, 5>
   }
// Set up the camera
viewpoint {
   from <0,5,-15>
   at <0,2,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0

   }

directional_light <0, 3, -10>
directional_light <0, 3, 4>

object {
   sphere <0,1.5,0>, 1.5
   Granite
}
object {
   sphere <-3.5,1.5,0>, 1.5
   Granite
}
object {
   sphere <3.5,1.5,0>, 1.5
   Granite
}

object {
   Plane_XZ
   texture { checker matte_white, matte_black }
}

object {
   Plane_XZ
   rotate <90, 0, 0>
   translate<0,0, 10>
   Sky
}
```

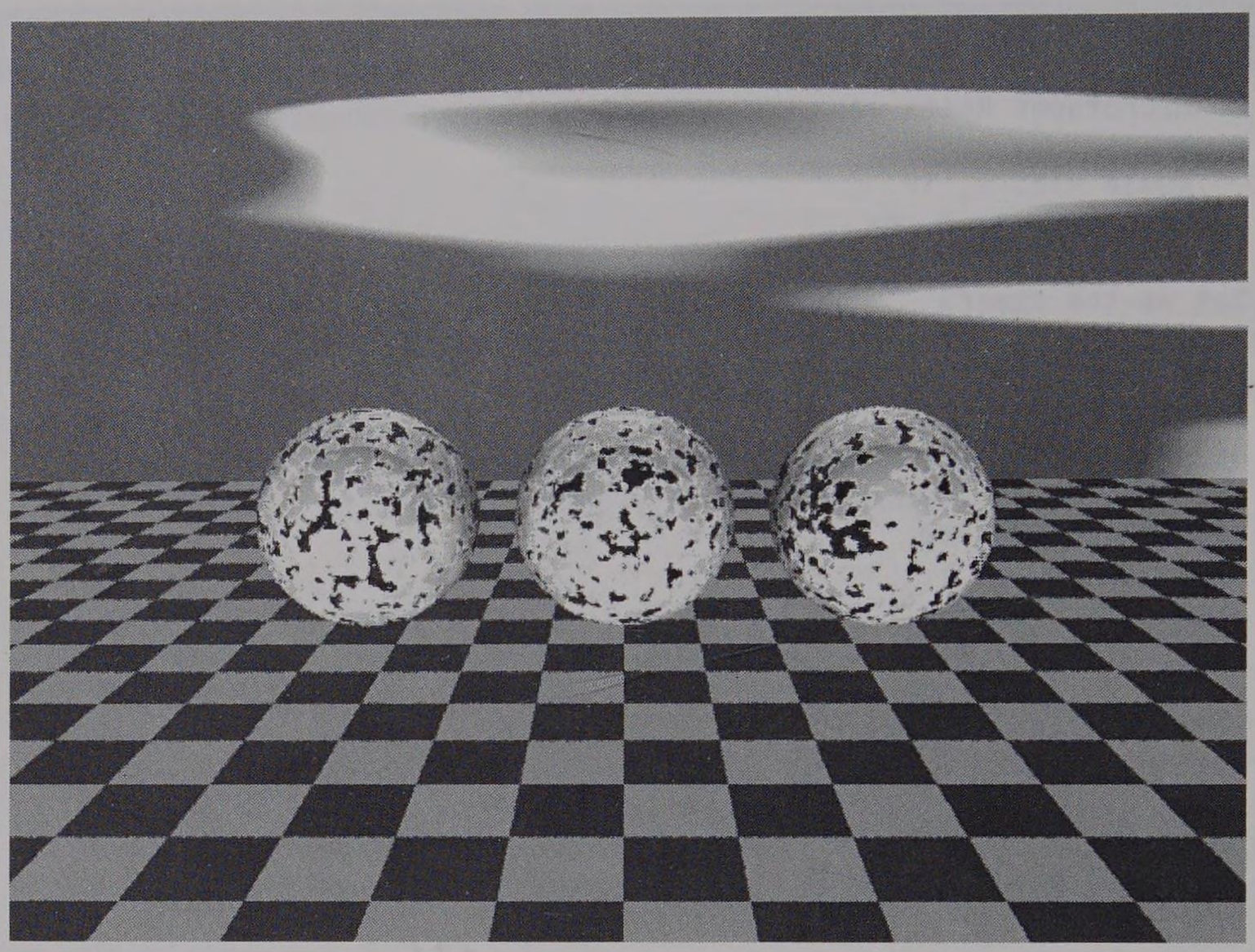


Figure 4.19

Functional textures Granite and Bozo (SPECIAL.PI).

Image Mapping Textures

Functional textures support the use of images as image maps. The image can be projected to a surface in four ways: planar, cylindrical, spherical, and environment. The syntaxes for the four types of projection are

```
planar_imagemap(myimage,coordinate[,repeat])
cylindrical_imagemap(myimage,coordinate[,repeat])
spherical_imagemap(myimage,coordinate)
environment_map(coordinate,environment("myfile1.tga","myfile2.tga",
                                       "myfile3.tga","myfile4.tga",
                                       "myfile5.tga","myfile6.tga"))
```

where *myimage* is defined as

```
define  myimage image("myfile.tga")
```

in which *coordinate* is a vector `<x,y,z>` used to select a color. The value of `x` is multiplied by the number of columns and `z` is multiplied by the number of rows. The color found at the new position in the image is used. The suggested vector is `P`. `P` is a reserved symbol for the position of the intersection between the light ray and the surface to be colored. In the environment map, the vector `reflect(I,N)` is suggested. The vector `I` defines the direction of the light ray that intersected the object, and `N` is the normal of the surface at the intersection point. The function `reflect(I,N)` then rotates the direction of `I` so that it is aligned with `N`. Polyray has various internal variables defined at run time; the variable `P` is a vector that represents the current intersection point between a light ray and the object. The vector `P` is normally used as the `coordinate` parameter, for example:

```
define myimage image("clown.tga")
planar_imagemap(myimage,P)
```

Chapter 6," Additional Features," explains Polyray's runtime variable `P` and others.

repeat stands for a value that indicates that the image pattern is repeated in the same object. The image can be 8-, 16-, 24- or 32-bit uncompressed, RLE compressed, or color-mapped Targa.

In planar projections, the image is mapped in x between 0 and 1 and in z between 0 and 1. In other words, the image is perpendicular to the y axis, and resting on the xz plane, with the lower left corner at the origin. The cylindrical mapping takes place on a cylinder with one end at the origin and the other end at <0,1,0>. If the pattern is repeated, the copies are added along the y axis. The spherical projection uses a sphere at the origin. The top edge of the image will be located in the north pole and the bottom edge at the south. Finally, the environment projection maps six images around the surface (top, bottom, left, right, front, and back). On all projections, any part of the object that is not covered by the image is given the color of the pixel (0,0).

Listing 4.13 uses all four properties with the same image file that was used to create the height fields (PLASMA.TGA). See figure 4.20 for the output of Listing 4.13.

Listing 4.13. Image mapping (MAPS.PI).

```
// MAPS.PI

include "texture.inc"
include "colors.inc"

define Plane_XZ  object {
            polygon 4, <0,0,0>, <1,0,0>, <1,0,1>, <0,0,1>
       }

define myimage image("plasma.tga")
define CImage
  texture {
     special surface {
      color cylindrical_imagemap(myimage,P)
      ambient 0.9
      diffuse 0.1
     }
   }

define SImage
  texture {
```

continues

Listing 4.13. Continued.

```
      special surface {
       color spherical_imagemap(myimage,P)
       ambient 0.9
       diffuse 0.1
      }
   }

define PImage
  texture {
     special surface {
      color planar_imagemap(myimage,P)
      ambient 0.9
      diffuse 0.1
     }
   }

define EImage
  texture {
     special surface {
      color environment_map(reflect(I, N),
                            environment("plasma.tga","plasma.tga",
                                        "plasma.tga","plasma.tga",
                                        "plasma.tga","plasma.tga"))
      ambient 0.9
      diffuse 0.1
     }
   }


// Set up the camera
viewpoint {
   from <0,5,-10>
   at <0,4,0>
   up <0,1,0>
   angle 45
   resolution 200,200
   aspect 1.0
   }

background <1, 1, 1>
light <0, 3, -10>

object {
   sphere <0,2,0>, 1.5
   SImage
}

object {
   sphere <0,6,0>, 1.5
   EImage
}
```

```
object {
  cylinder <0, 0, 0>, <0, 1, 0>, 1
  CImage
  scale <1,3,1>
  rotate <-20, 10, 0>
  translate <3.5, 2, 2>
 }

object {
   Plane_XZ
   PImage
   scale <5,5,5>
   rotate <90, -30, 0>
   translate<-10,3, 15>
}
```

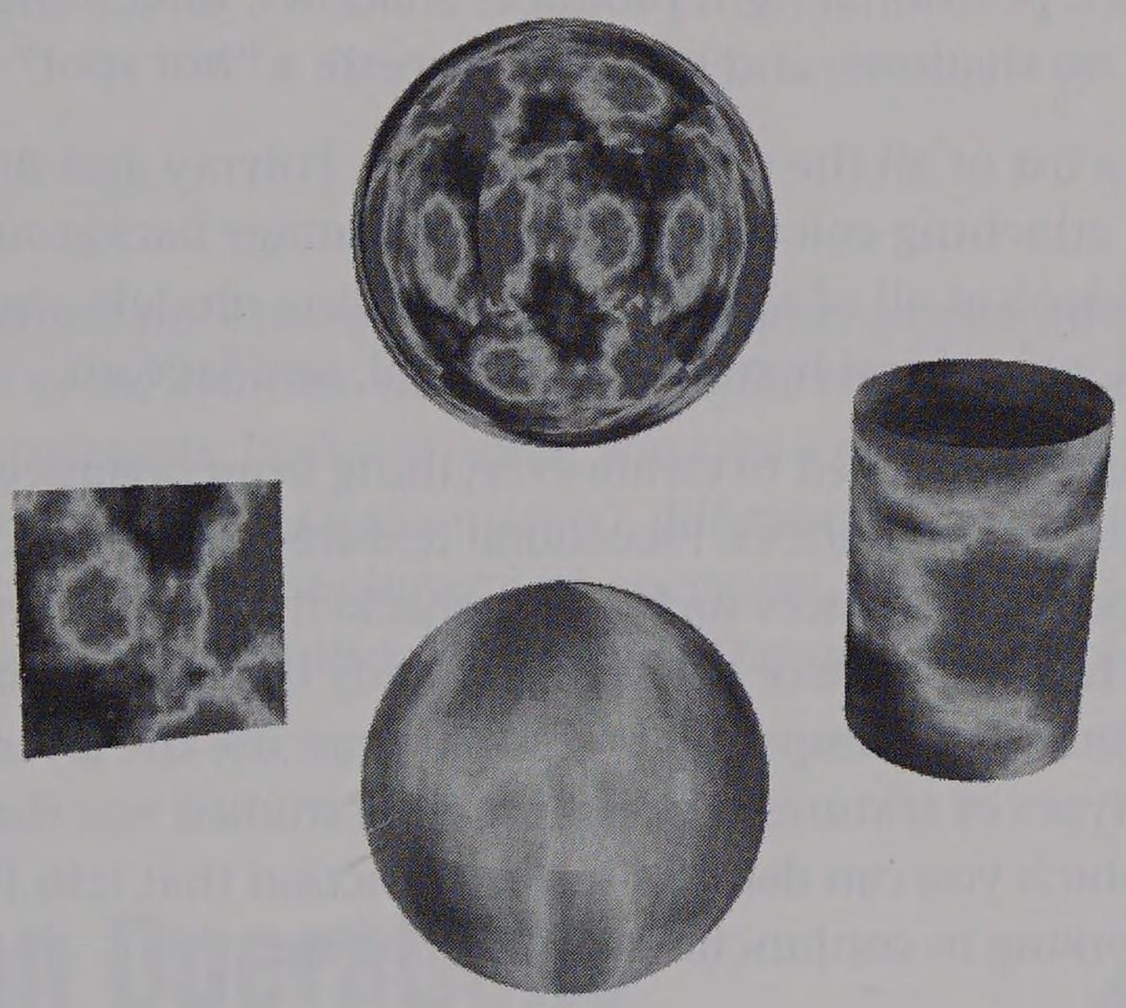


Figure 4.20

Mapping an image file to several surfaces: sphere, cylinder, and plane (MAPS.PI).

Scaling and translation are very important when you're using image maps. For example, the cylinder is initially the same size and in the same location as the mapping procedure requires. To scale it appropriately, you need to apply the transformation after the texture declaration, as in the following:

```
object {
  cylinder <0, 0, 0>, <0, 1, 0>, 1
  CImage
  scale <1,3,1>
  rotate <-20, 10, 0>
  translate <3.5, 2, 2>
}
```

If the object isn't created in the place and size required by the mapping function, the object might not receive the mapping image at all.

This concludes the texture discussion. In Chapter 5, we cover some miscellaneous features, and you look at Polyray's mathematical computation capabilities. If you're not a math-oriented person, you may want to skip the last section of Chapter 5, "Constructive Solid Geometry."

Summary

In this chapter, you learned the different types of light sources: the positional, the directional, and the spot light. Each light source helps you create a different effect: positional light produces shadows, directionals produce only light and no shadows, and spot lights create a "hot spot" on the image.

Next, you saw a list of all the colors available in Polyray and different techniques for attaching colors to surfaces and image backgrounds. You also took a detailed look at all of the different texturing models: standard shading model, procedural textures, functional, and layered.

The standard model was used to create everything from opaque surfaces to transparent surfaces and mirrors. Procedural textures are ready-to-use patterns that were mapped into surfaces such as the checker, the hexagon, and the noise surface. The noise surface texture was handy in creating marble, ripples in water, and dents and bumps in surfaces. You can use the layered texture to stack different types of textures. The next texture studied was the functional texture, with which you can define a lookup function that tells Polyray what color to use, working in conjunction with color maps.

Finally, you learned how to map images to different surfaces by means of image maps. The four kinds of image maps are planar, cylindrical, spherical, and environment.

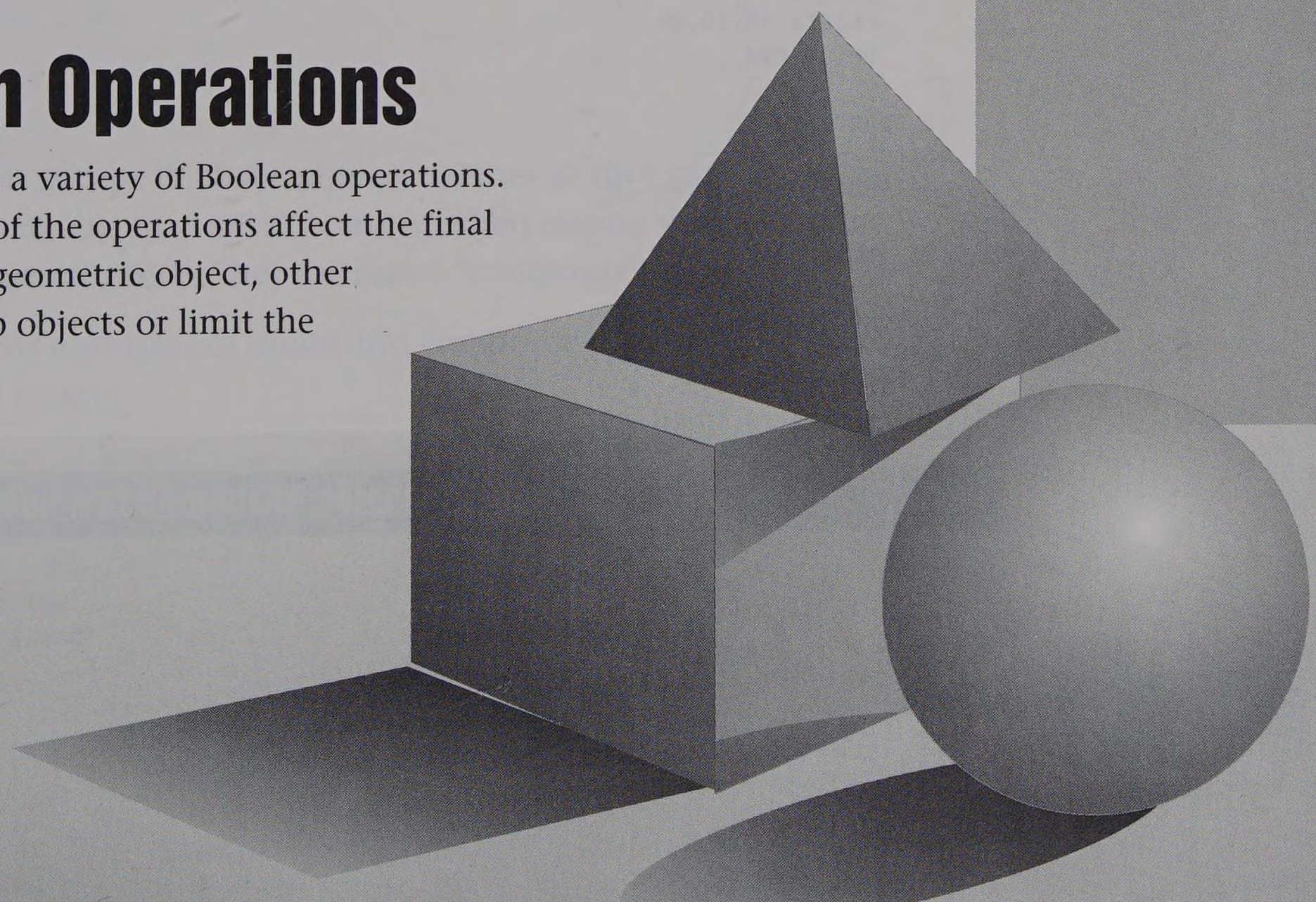
In the next chapter, you take a look at constructive solid geometry and how Polyray handles those operations.

Constructive Solid Geometry

Chapter 1 presented the concept of creating complex objects with simple primitives such as the cylinder, the sphere, and the box, in conjunction with Boolean operations. In this chapter, you take a practical approach at using Boolean operations such as union (+), difference (-), and intersection (*). In addition, you learn the clipping (&) operation, the inverse (~) of an object, and how to define artificial boundaries through the use of a bounding box.

Boolean Operations

Polyray supports a variety of Boolean operations. Although some of the operations affect the final form of a given geometric object, other operations group objects or limit the



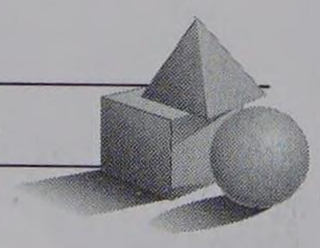
extension of an object by clipping any part that is outside a defined boundary. Grouping objects is an extremely powerful feature you can use to manipulate a group of objects as a whole instead of having to apply transformations to each object. For example, a car consists of a group of objects: the tires (tori) and the chassis (box). By grouping them together, you can translate the *set* of objects (the car) without having to translate each tire and each section of the chassis.

Union

Unions in Polyray are represented by the symbol +. When a union operation is performed on a set of shapes or objects, the resulting set itself becomes a single object consisting of all the individual shapes. The new object can be manipulated by means of transformations or other Boolean operations. For example, assume you have a set of four spheres on the xy plane and wish to rotate them as a group, around the y axis by 15 degrees. You can do this in two ways:

Method #1

```
// sphere #1
object {
    sphere <1,1,0> ,2
     rotate <0,15,0>
     shiny_red
}
// sphere #2
object {
    sphere <-1,1,0> ,2
     rotate <0,15,0>
     shiny_red
}
// sphere #3
object {
    sphere <-1,-1,0> ,2
     rotate <0,15,0>
     shiny_red
}
// sphere #4
object {
    sphere <1,-1,0> ,2
     rotate <0,15,0>
     shiny_red
}
```



Method #2

```
object {
   object {sphere <1,1,0>, 2}
+ object {sphere <-1,1,0>, 2}
+ object {sphere <-1,-1,0>, 2}
+ object {sphere <1,-1,0>, 2}
     rotate <0,15,0>
     shiny_red
}
```

As you can see, you greatly reduced the amount of typing by using the union (+) operator. Only one rotation and one texture statement were needed for the entire group. You can still apply transformations to individual objects as before; you can replace the statement

```
object { sphere <1,1,0>,2}
```

with

```
object {sphere <0,0,0>,2
translate <1,1,0>
}
```

giving the same result. Sometimes you have to control a group of objects as a whole, even though each object in the corresponding group has a different texture. Take the following set of spheres, for example:

```
object {
     object { sphere <0,1,0>,2
              shiny_red
     }
+    object { sphere <0,-1,0>,2
              shiny_green
     }
     translate <1,1,5>
}
```

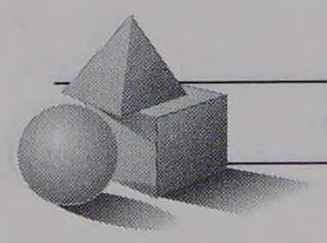
A union operation groups the two spheres together so they can be translated with just one statement. In some ray tracers, when objects with different textures are grouped together, they are called "composite" objects.

Listing 5.1 presents examples of union and composite objects, and figure 5.1 displays the final image.

Listing 5.1. Union and composite objects (UNION.PI).

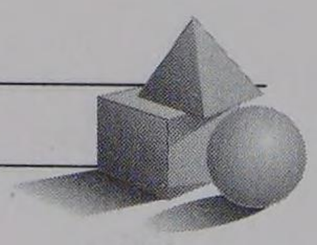
```
//UNION.PI
include "colors.inc"
include "texture.inc"
```

continues



Listing 5.1. Continued.

```
viewpoint {
 from      <0.2, 4.0, -5.0 >
 at        <0.0, -1.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
  light < 10 ,  10 , -10  >
// LIGHT_SOURCE
  light <-10 ,  10 , -10  >

define TwoSpheres
object {
  object { sphere <  0.5, 0, 0>, 0.47  }
+ object { sphere < -0.5, 0, 0>, 0.47  }
}
define GraySpheresUnion
object {
  object {
    TwoSpheres
  }
+  object {
    TwoSpheres
    rotate<0, 0, 90>
  }
+ object {
    TwoSpheres
    rotate<0, 90, 0>
  }
  reflective_gray
}
define GraySpheresComposite
object {
  object {
    TwoSpheres
    matte_red
  }
+  object {
    TwoSpheres
    rotate<0, 0, 90>
    matte_green
  }
+ object {
    TwoSpheres
    rotate<0, 90, 0>
    matte_yellow
  }
}
object {
   GraySpheresUnion
   translate<-1.5, 0, 0>
}
object {
   GraySpheresComposite
```



```
      translate<1.5, 0, 0>
   }
   object {
      object {
         cone <0.0, -1.3, 0.0>, 0.8, <0.0, 0.0, 0.0>, 0.0
         translate<-0.2, -1.5, 0>
         reflective_red
      }

+     object {
         cone <0.0, -1.3, 0.0>, 0.8, <0.0, 0.0, 0.0>, 0.0
         translate<0.2, -1.5, 0>
         reflective_gray
      }
   }
```

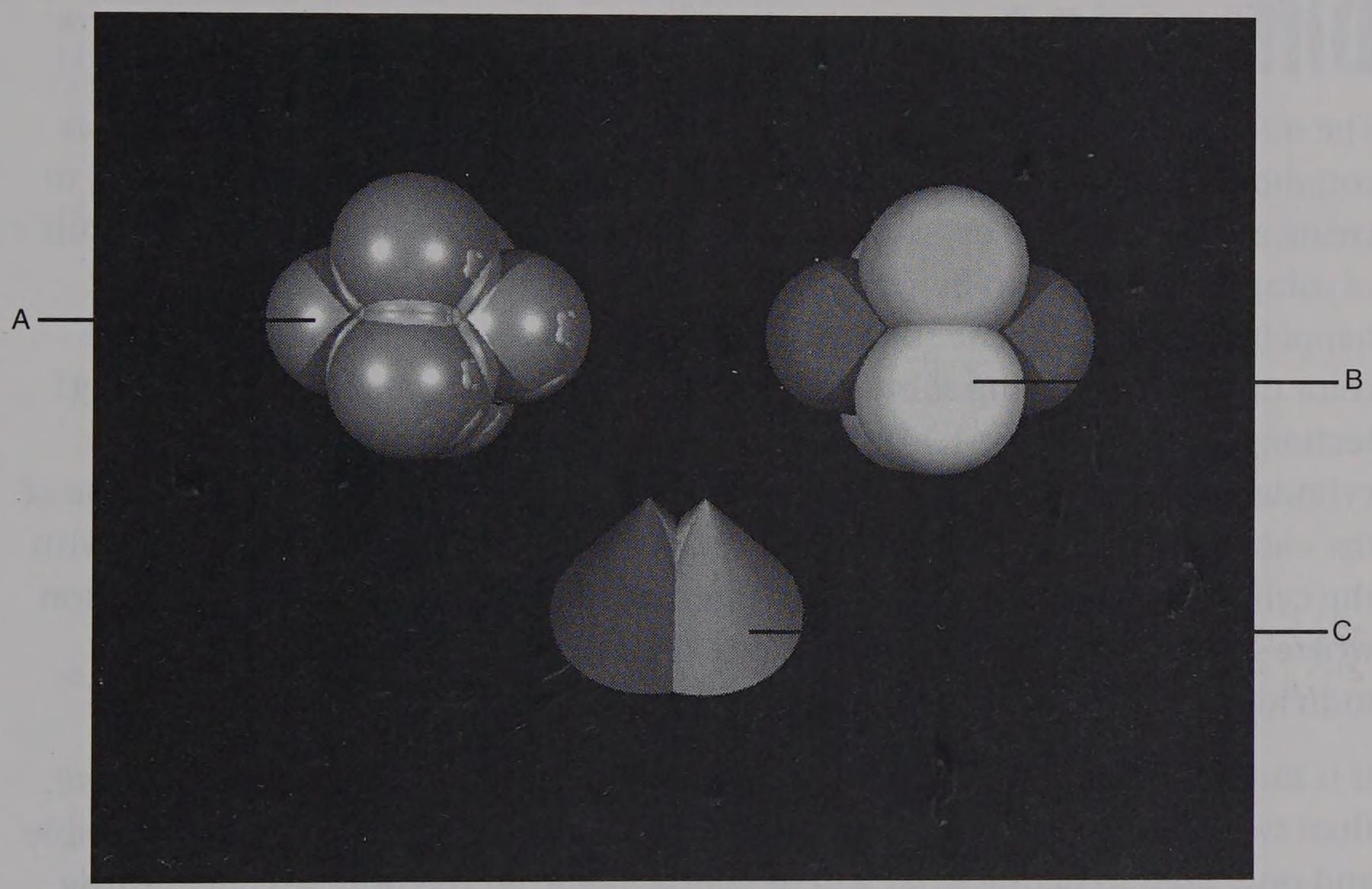


Figure 5.1

Using unions and composite objects (UNION.PI).

In the UNION.PI scene, initially two spheres are grouped together and called `TwoSpheres`. This new object (A) becomes the base object you will use to create a more complex object. The object `GraySpheresUnion` is composed of three copies of the `TwoSpheres` primitive. The second and third objects (B and C) are rotated about Z and Y by 90 degrees, respectively. The color used is `reflective_gray`. Using the same primitive, the `GraySpheresComposite` object is created by attaching individual textures to each shape and the appropriate transformation. The statements

```
object {
   GraySpheresUnion
   translate<-1.5, 0, 0>
}
object {
   GraySpheresComposite
   translate<1.5, 0, 0>
}
```

are used to locate the objects in their final position. Finally, two cones are created as a composite and located underneath `GraySpheresComposite` and `GraySpheresUnion`. Observe that even though the cones are intersecting each other, their individual colors show the limits of each one.

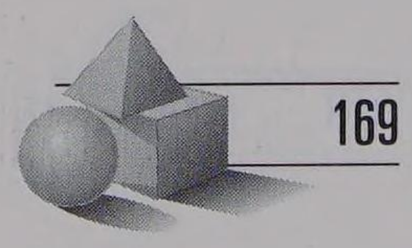
Difference

The difference (-) operation is used to subtract one object from another and is commonly used to create holes in shapes. In figure 1.11, a cylinder was used to create a hole in a cube by using the difference operation. One interesting result of using differences can be seen if the colors of the objects are different (as it happens in fig. 1.11). The surface where both objects are in contact takes the color of the object being subtracted, thus giving the appearance of a "missing" section. Again, in figure 1.11, the cube has a reflective gray color while the cylinder has a reflective red surface. After applying the difference, the volume of the cube that was inside the cylinder disappears and the surface in contact with the cylinder becomes reflective red. This method relates to the previous section where some of the objects had only one texture while another group had individual textures for each object.

It is important to note that when a difference operation is performed on more than two objects, the first object becomes the primary, and the rest are invisible and only used to define what will be subtracted from the primary one. Listing 5.2 uses a cube to represent the primary object. Two torus objects will be subtracted from the cube. In addition, a second example in the same image also is presented. A sphere is used as the primary object from which two more tori will be subtracted. Figures 5.2 and 5.3 present the objects without and with the difference operations, respectively. To generate figure 5.2, change the difference operators (-) with union operators (+).

Listing 5.2. Two difference operations (DIFFR.PI).

```
//DIFFR.PI
include "colors.inc"
include "texture.inc"
viewpoint {
 from      <1, 1.0, -7.0 >
 at        <1.2, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
  light < 10 ,  10 , -10  >
// LIGHT_SOURCE
  light <-10 ,  10 , -10  >

object {
// BOX
    object {
          box <-1, -2, -1>, <1, 2, 1>
    }
// TORUS
-   object {
      torus  1.5, 0.6, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
      translate <0, 1, 0>
    }
// TORUS
-   object {
      torus  1.5, 0.6, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    }
    rotate<0, 15, 0>
    reflective_red
    translate<-0.5, 0.1, 0>

object {
   object {
      sphere <0, 0, 0>, 1.2
      matte_green
    }
-   object {
      torus  1.1, 0.3, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
      matte_blue
    }
-   object {
      torus  1.1, 0.3, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
      rotate<0, 0 ,90>
      matte_red
    }

  rotate<15, 0, 0>
  translate <3, 0, 0>
}
```



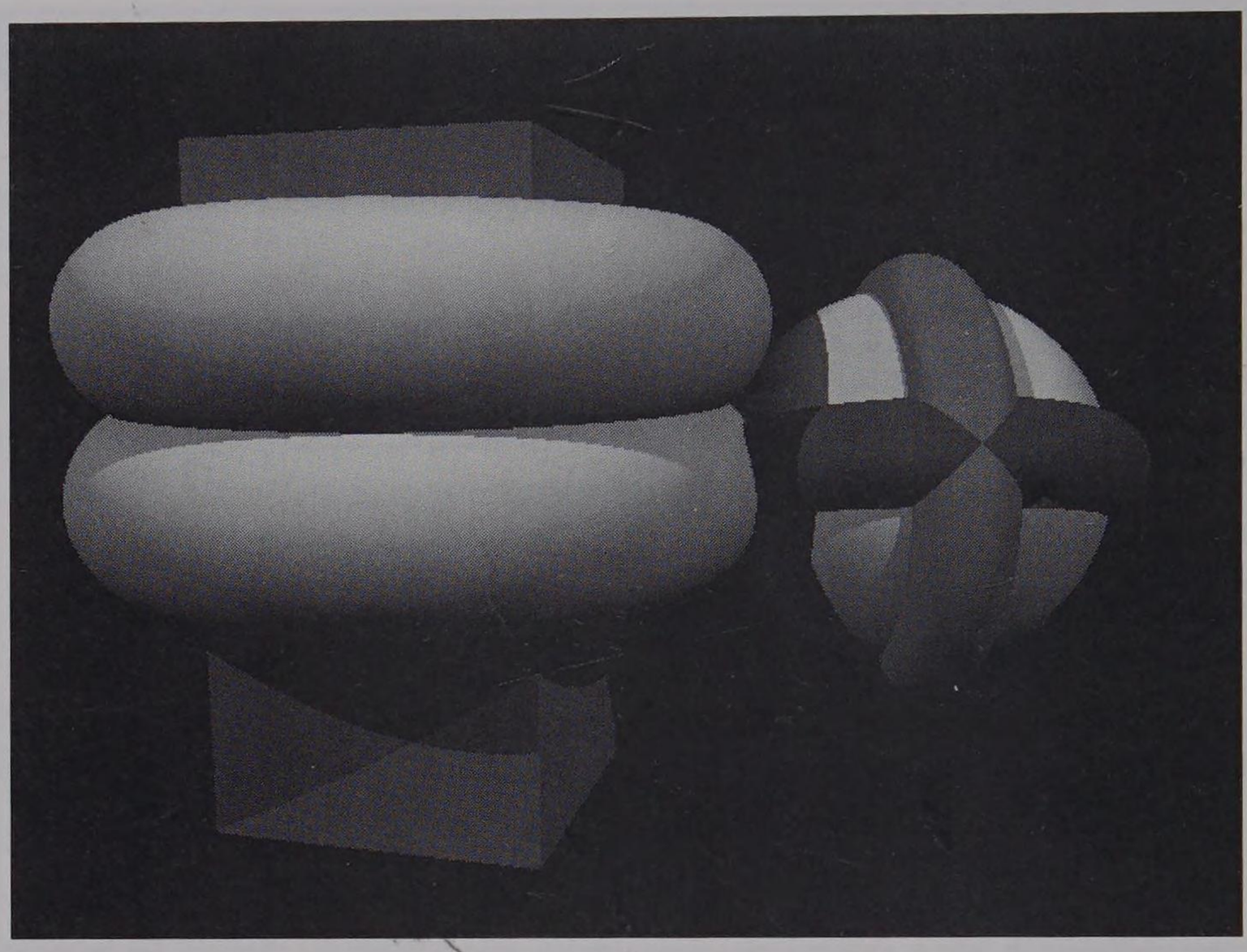


Figure 5.2

Listing 5.2 without the difference operations.

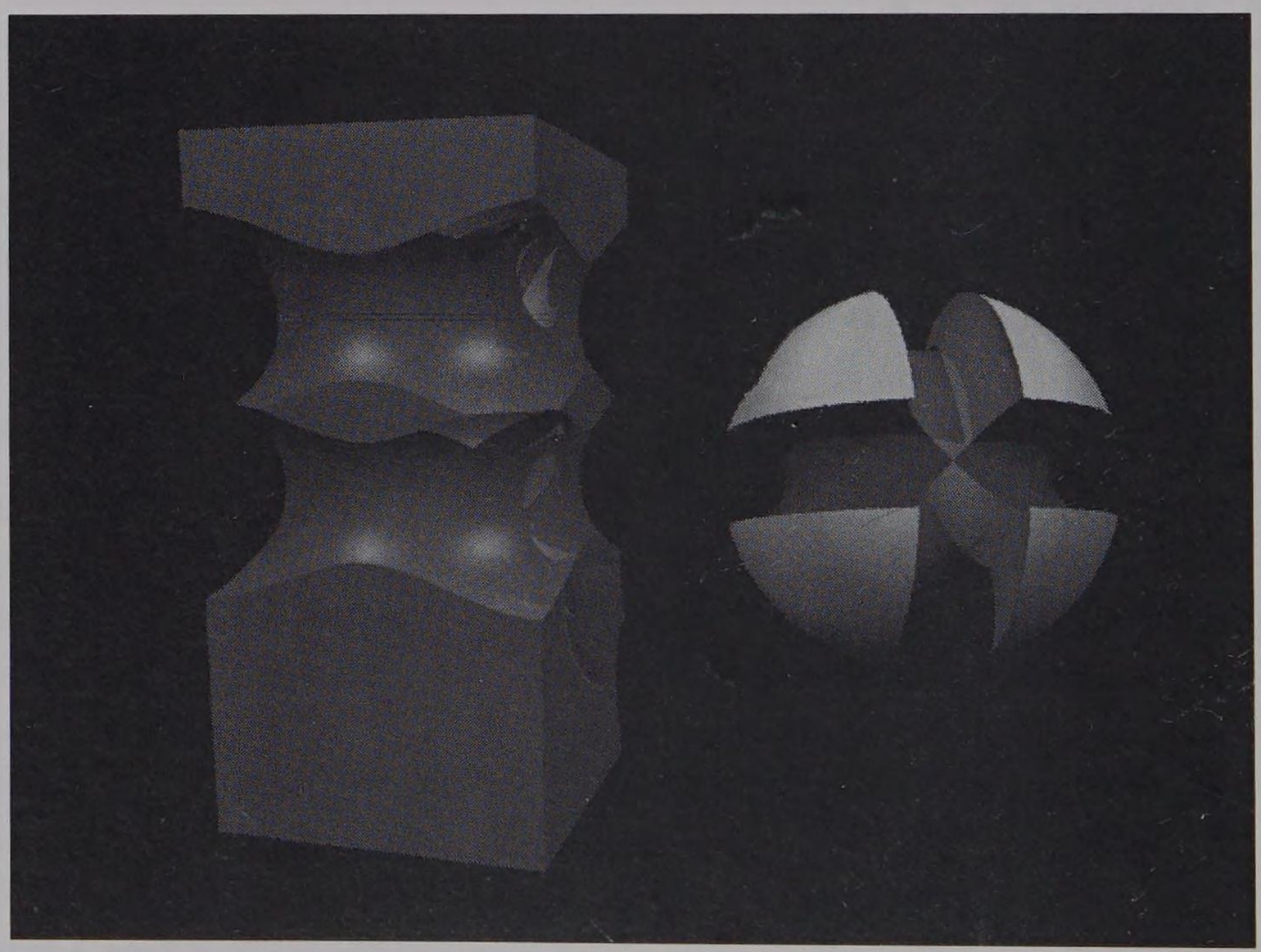


Figure 5.3

Listing 5.2 with the difference operations (DIFFR.PI).

Observe in figure 5.3 how the sphere's surfaces that were in contact with the tori have taken their respective colors. This feature accentuates the grooves on the sphere.

Intersection

Intersections (*) are easy to use but sometimes difficult to visualize. An intersection operation creates an object that consists of all the surface sections of each object which are inside one another, or overlapping. In Listing 4.3, an intersection was used to create the neck of the lamp. The code used was

```
// LAMP "NECK"
object  {
// TORUS
object {
    torus  0.4, 0.05, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate <90, 0, 0>
    translate <-0.375, 0.3, 0>
}
*
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale <0.4, 0.4, 0.4>
    translate <0.125, 0.575, 0>

}
 LampColor
}
```

As you can see, the order of the objects is not important. The box was used as an artificial boundary. Any part of the torus that was outside of the box was cut. The next section covers the clipping operation, which you can use for the same purpose.

Listing 5.3 presents an intersection operation.

Listing 5.3. Intersection operation (INTERS.PI).

```
//INTERS.PI
include "colors.inc"
include "texture.inc"
define ColoredGlass
   surface {
      ambient 0.3
      diffuse 0
      specular 0.2
      reflection white, 0.1
      transmission white, 0.7, 1
      }
define RedGlass texture {ColoredGlass {color red}}

viewpoint {
 from      <1, 1.0, -7.0 >
 at        <0.0, -1.5, 0.0 >
```

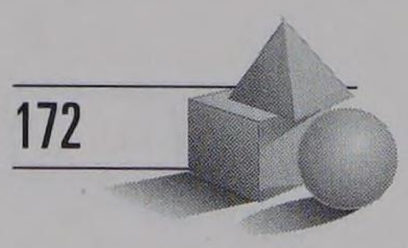
continues

Listing 5.3. Continued.

```
 up         <0.0, 1.0, 0.0 >
 angle       45
 resolution 100,100
 aspect      1.0
}
// LIGHT_SOURCE
  light < 10 ,  10 , -10  >
// LIGHT_SOURCE
  light <-10 ,  10 , -10  >

object {
// BOX
    object {
        box <-1, -1, -1>, <1, 1, 1>
        rotate<-15, 0, 0>
        RedGlass
    }
+   object {
        sphere <1, 0, 0>, 1
        shiny_yellow
    }
    rotate<0, 15, 0>
}
object {
// BOX
    object {
        box <-1, -1, -1>, <1, 1, 1>
        rotate<-15, 0, 0>
    }
*   object {
        sphere <1, 0, 0>, 1
    }
    translate<0, -3, 0>
    rotate<0, 15, 0>
    shiny_yellow
}
```

There are two parts in Listing 5.3. The first part displays the box as a transparent red object and the sphere as yellow. Looking at both objects, you can see that the only part of the sphere that is inside the box is the left hemisphere. And only the part of the box that is inside the left hemisphere will remain after the intersection operation.

The second part of Listing 5.3 is the final intersection operation. The box has no texture, so it will only be used as an artificial boundary. The only part of the box that will be visible is the surface that covers the open area in the hemisphere.



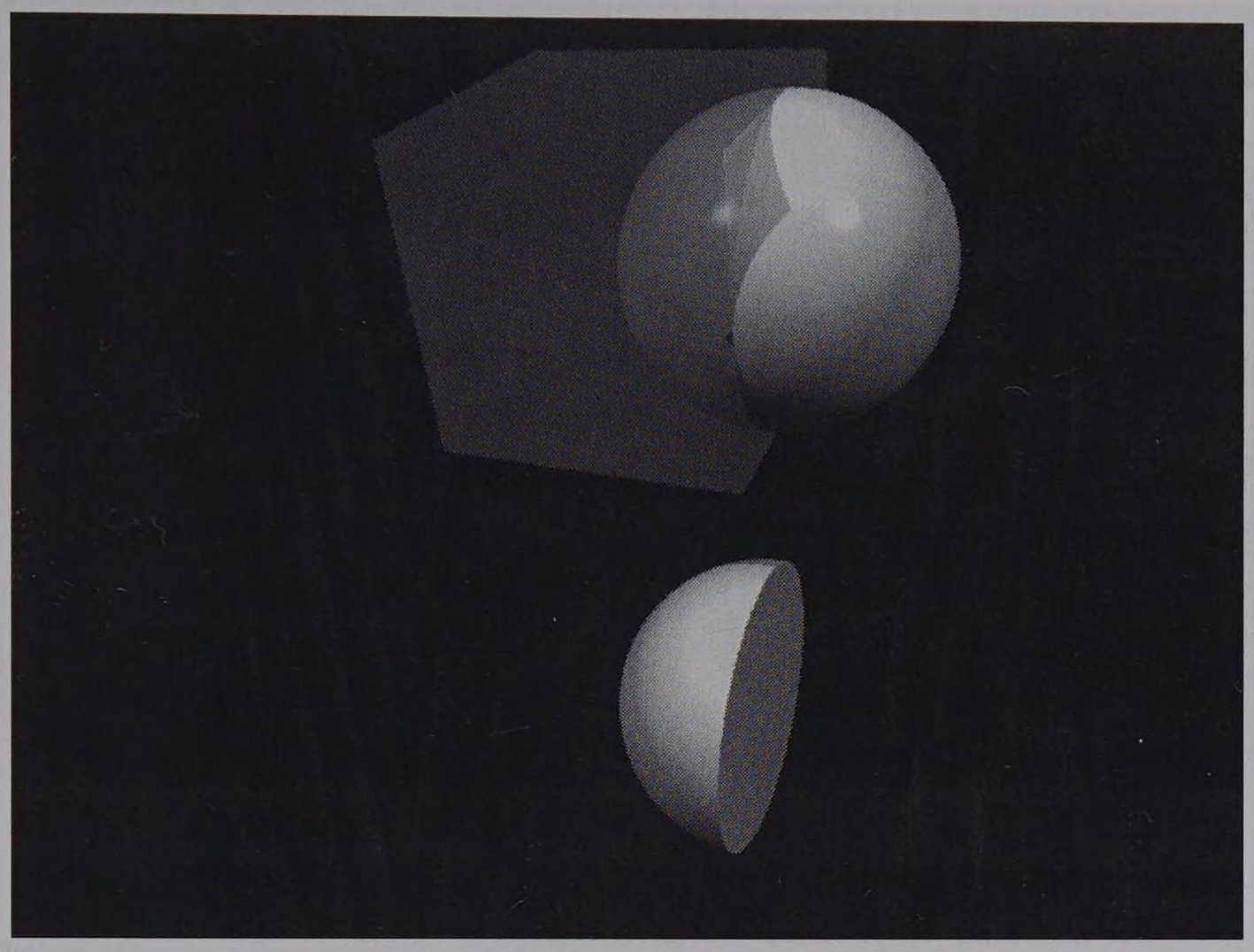


Figure 5.4

Example of intersection operation between a box and a sphere (INTERS.PI).

Clipping

Clipping an object can be accomplished by using the & operator. Any part of the primary object that goes beyond the clipping boundary (the clipping object) will not be seen. Let's use figure 5.4 as an example. If we had used a clipping operation instead of a difference, we would have the same results, except that the left hemisphere would appear hollow instead of solid. Listing 5.4 uses the same objects presented in Listing 5.3 (the box and the sphere) to demonstrate a clipping operation.

Listing 5.4. Clipping objects (CLIP.PI).

```
//CLIP.PI
include "colors.inc"
include "texture.inc"
define ColoredGlass
   surface {
      ambient 0.3
      diffuse 0
      specular 0.2
      reflection white, 0.1
      transmission white, 0.7, 1
      }
define RedGlass texture {ColoredGlass {color red}}

viewpoint {
 from      <1, 1.3, -7.0 >
```

continues

Listing 5.4. Continued.

```
 at         <0.0, -1.5, 0.0 >
 up         <0.0, 1.0, 0.0 >
 angle       45
 resolution 100,100
 aspect      1.0
}
// LIGHT_SOURCE
directional_light < 10 ,  10 , -10  >
// LIGHT_SOURCE
directional_light <-10 ,  10 , -10  >
background gray
object {
// BOX
    object {
        box <-1, -1, -1>, <1, 1, 1>
        RedGlass
    }
+   object {
        sphere <0, 0, 0>, 1.2
        shiny_yellow
    }
    rotate<0, 20, 0>
}
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
&   object {
        box <-1, -1, -1>, <1, 1, 1>
    }
    translate<0, -3, 0>
    rotate<0, 20, 0>
    shiny_yellow
}
```

Listing 5.4 follows the same approach used in the previous section. The box (the clipping object) is transparent, permitting the view of what is inside and what is outside the boundary. The sphere is the primary object, and since it is larger than the box, parts of it will be clipped. In figure 5.5 (the lower section), the sphere is clipped, appearing to be hollow. The clip operation does not close the surface.

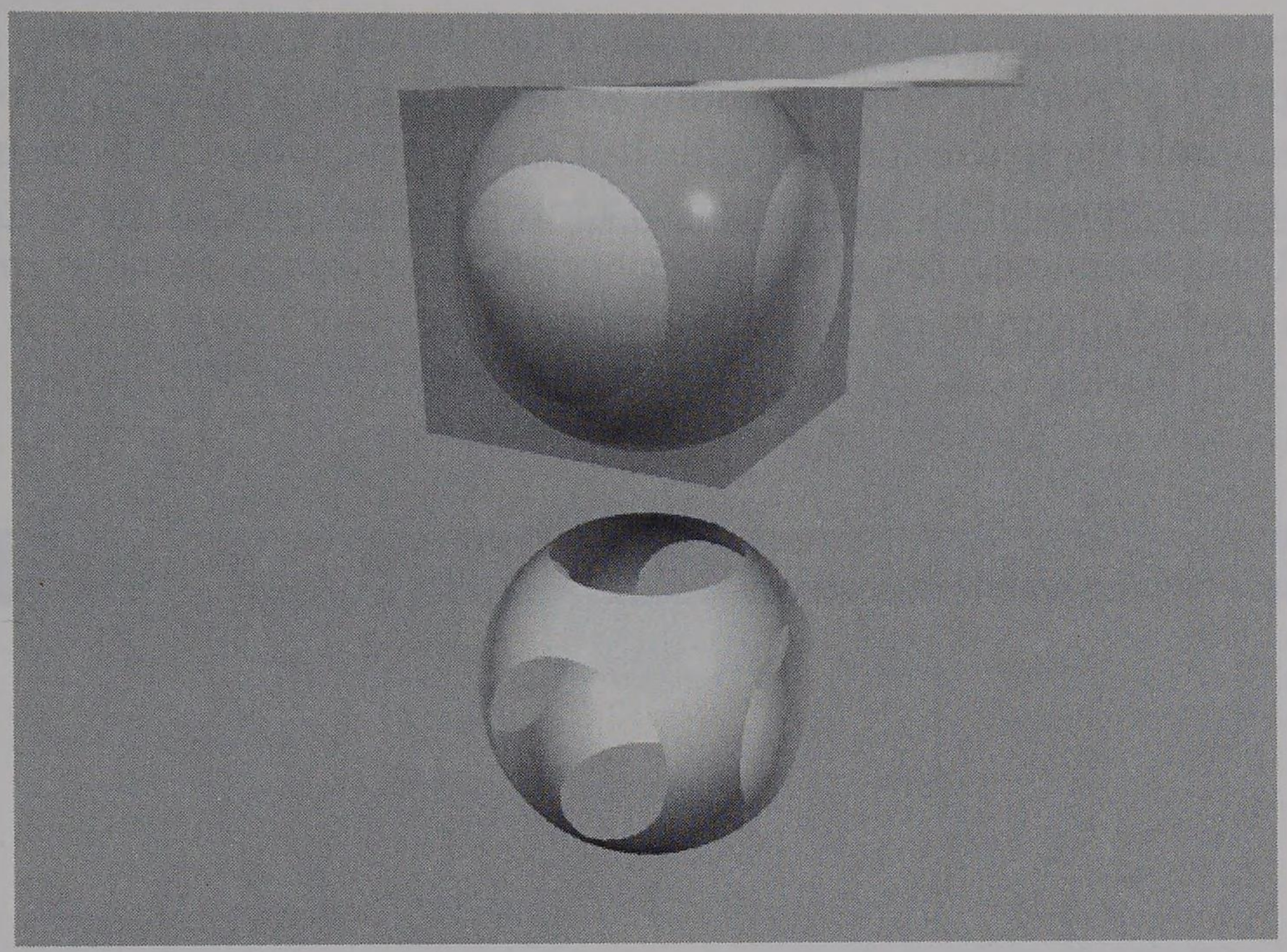


Figure 5.5

Clipping a sphere with a box (CLIP.PI).

Bounding Box

A bounding box defines the limits on an object or group of objects in order to speed up the ray tracing operations. Bounding boxes were covered briefly in Chapter 3, in the "Blob" section and later in the "Functions" section. This is a section of the code used in the scene BLOB.PI in Chapter 3:

```
object {
    object {
      blob 0.5:
      plane <0,1,0>, 0, 2, 5,
      sphere <0,3,0>, -4, 2
       RedFinish
    }
    bounding_box <-3,1,-1>, <3,3,3>
    scale <0.5,0.5,0.5>
}
```

In this section, we used a bounding box to limit the plane surface. A plane is an infinite flat surface. Since planes have no boundaries, one was created using the statement `bounding_box <-3,1,-1>, <3,3,3>`. This statement forces the ray tracer to check the plane only if the current light ray intersects (passes through) a box with the limits `<-3,1,-1>` and `<3,3,3>`. This feature speeds the ray tracing calculations since it is faster to check the bounding box than to check the blob object for its visibility.

Interestingly enough, bounding boxes can be used to clip an object (even though it is NOT recommended). Listing 5.5 uses a box and a sphere. The box is located inside the sphere and is roughly half as big as the sphere. The box is only used to represent the bounding box; a true box is not used in the operation. Using bounding boxes as a clipping operation is risky and prone to errors. If you need to clip an object use the appropriate operation. This example is given only to present the similarities between bounding box and clipping. Figure 5.6 shows the output of Listing 5.5.

Listing 5.5. Using bounding boxes to clip objects (BOUNDED.PI).

```
//BOUNDED.PI
include "colors.inc"
include "texture.inc"
define ColoredGlass
   surface {
      ambient 0.3
      diffuse 0
      specular 0.2
      reflection white, 0.1
      transmission white, 0.7, 1
      }
define YellowGlass texture {ColoredGlass {color yellow}}
viewpoint {
 from      <1, 1.3, -7.0 >
 at        <0.0, -1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
directional_light < 10 ,  10 , -10  >
// LIGHT_SOURCE
directional_light <-10 ,  10 , -10  >
background gray
object {
    object {
        box <-0.5, -0.5, -0.5>, <0.5, 0.5, 0.5>
        texture { surface {color red}}
    }
+   object {
        sphere <0, 0, 0>, 1.2
        YellowGlass
    }
    rotate<0, 20, 0>
}
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
```

```
    bounding_box <-0.5, -0.5, -0.5>, <0.5, 0.5, 0.5>
    translate<0, -3, 0>
    rotate<0, 20, 0>
    shiny_yellow
}
```

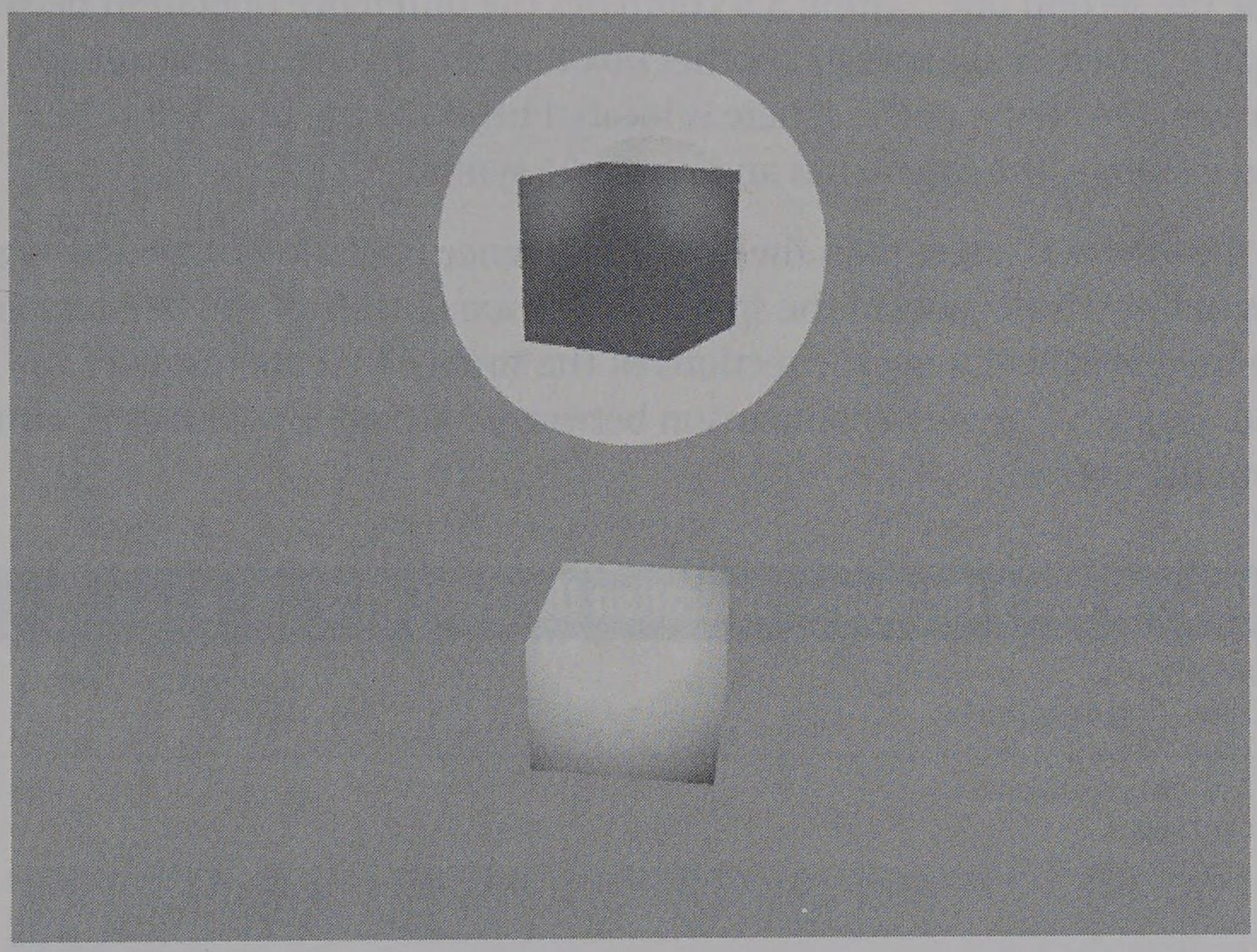


Figure 5.6

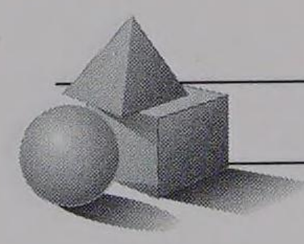
Clipping objects with bounding boxes (BOUNDED.PI).

The section that clips the sphere is

```
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
    bounding_box <-0.5, -0.5, -0.5>, <0.5, 0.5, 0.5>
    translate<0, -3, 0>
    rotate<0, 20, 0>
    shiny_yellow
}
```

The bounding box object tells the ray tracer not to look beyond the limits `<-0.5, -0.5, -0.5>`, and `<0.5, 0.5, 0.5>`, even though the sphere is larger than this range.

Bounding boxes are used to speed up ray tracing calculation in objects such as a collection of polygons. If you have an object consisting of 1,000 polygons, each time the ray tracer has to evaluate the polygons it will take a long time looking at each polygon. If a bounding box is used to define the limits, the ray tracer will only check one box. If it doesn't see the box, it doesn't need to check the polygons. Compare that to checking 1,000 polygons!



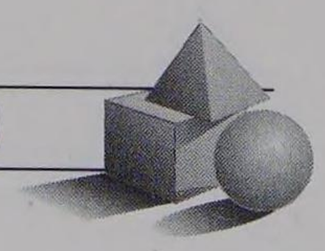
Inverse

The inverse (~) operation specifies what part of an object (the inside or outside) is important. If a given object is located inside another object, by attaching the inverse symbol (~) to the first object, the ray tracer assumes that the object is outside the second one. Listing 5.6 compares the difference operation between a box and a sphere and between another box and the inverse of that sphere. In both cases, the center of the sphere is located inside of the box, but in the second example, the sphere has an inverse (~) symbol.

Since the sphere is larger than the box, a difference operation of `SPHERE - BOX` results in only those parts of the sphere which are outside of the box remaining. In the inverse sphere, only the sections of the surface that are inside of the box remain. Figure 5.7 shows a comparison between a difference operation with and without the inverse.

Listing 5.6. Using the inverse operation (INVERSE.PI).

```
//INVERSE.PI
include "colors.inc"
include "texture.inc"
define ColoredGlass
   surface {
      ambient 0.3
      diffuse 0
      specular 0.2
      reflection white, 0.1
      transmission white, 0.7, 1
      }
define RedGlass texture {ColoredGlass {color red}}

viewpoint {
 from      <1, 1.3, -8.0 >
 at        <0.0, -1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
directional_light < 10 ,  10 , -10  >
// LIGHT_SOURCE
directional_light <-10 ,  10 , -10  >
background gray
object {
// BOX
    object {
        box <-1, -1, -1>, <1, 1, 1>
        RedGlass
    }
+   object {
```



```
        sphere <0, 0, 0>, 1.2
        shiny_yellow
    }
    rotate<0, 20, 0>
}
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
-   object {
        box <-1, -1, -1>, <1, 1, 1>
    }
    translate<-2, -3, 0>
    rotate<0, 20, 0>
    shiny_yellow
}
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
-   ~object {
        box <-1, -1, -1>, <1, 1, 1>
    }
    translate<2, -3, 0>
    rotate<0, 35, 0>
    shiny_yellow
}
```

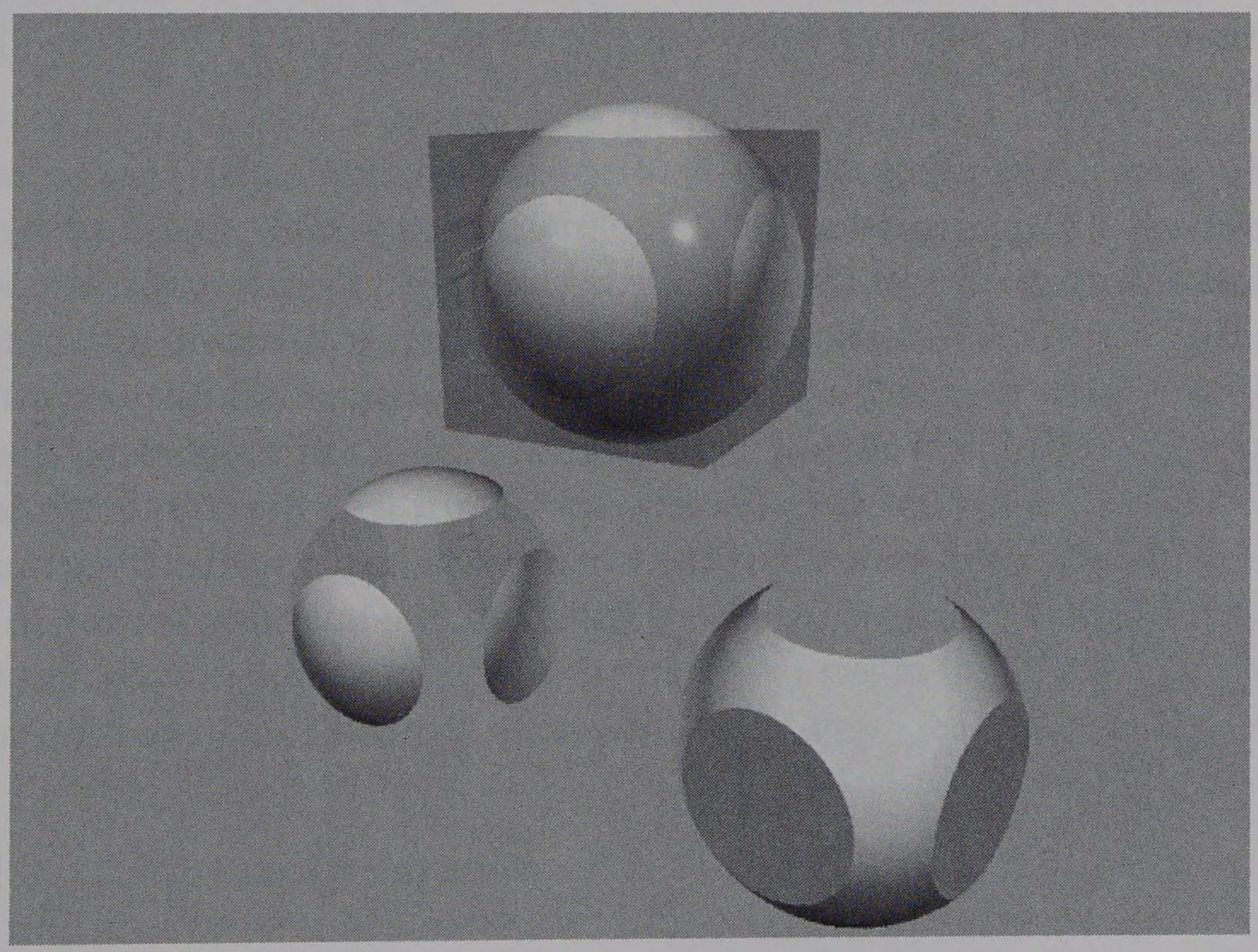


Figure 5.7

Comparison between a difference operation with and without the inverse (INVERSE.PI).

The section of code that applies the difference to the inverse of the box is

```
object {
    object {
        sphere <0, 0, 0>, 1.2
    }
-   ~object {
        box <-1, -1, -1>, <1, 1, 1>
    }
    translate<2, -3, 0>
    rotate<0, 35, 0>
    shiny_yellow
}
```

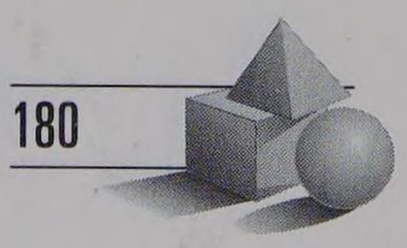
Place the ~ symbol in front of the object to invert.

Summary

In this chapter, you took a more detailed look at Boolean operations and how they are applied in Polyray. The union (+), difference (-), and intersection (*) operations are the most commonly used, and the syntax is similar. When you apply difference operations, the order of the objects is important. The first object becomes the primary object, and any additional objects will be used to subtract from the primary. The union and intersection operations do not have that requirement. Composite objects are unions in which each shape or object has its own texture.

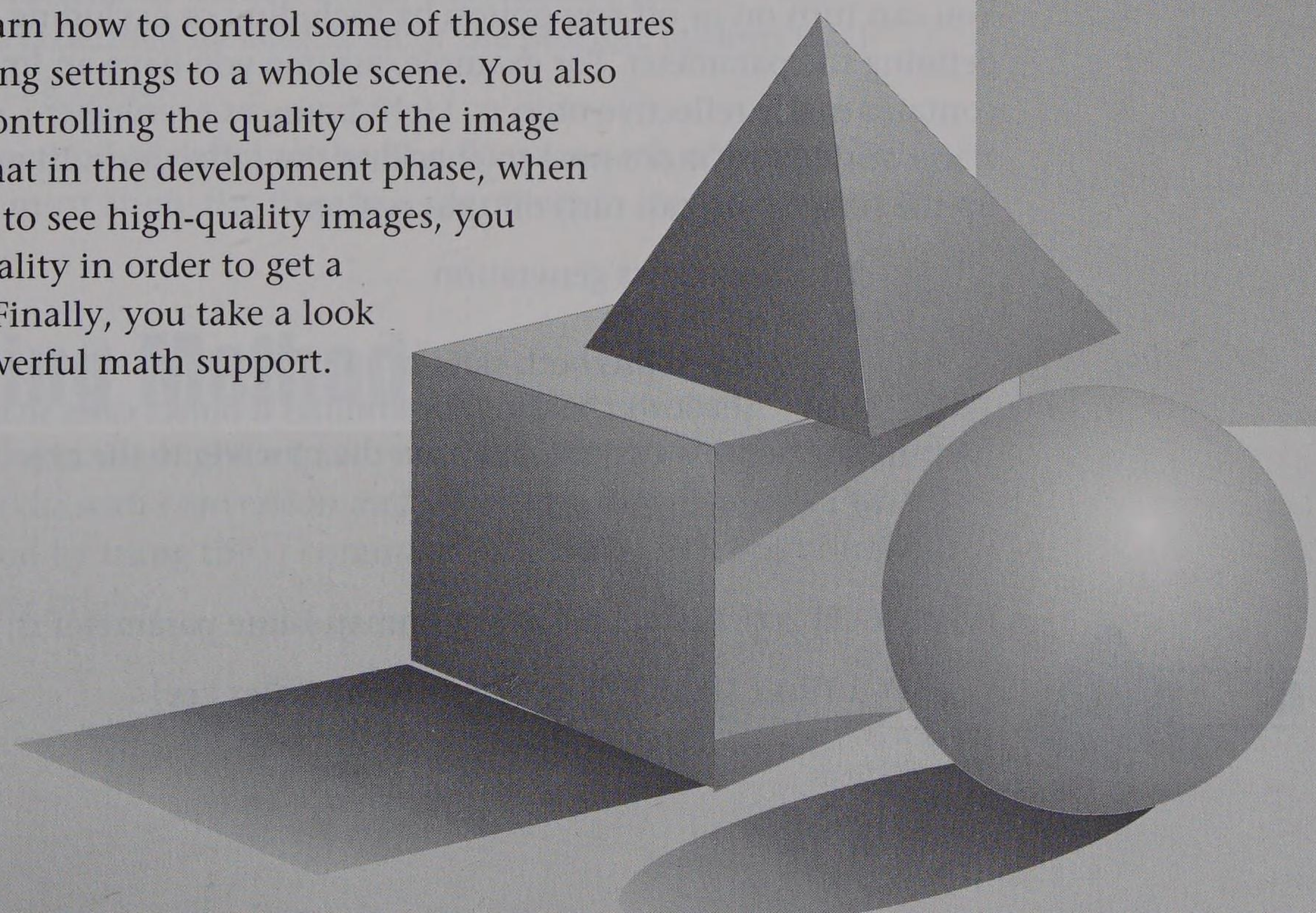
You also were introduced to a new set of operations: clipping (&), `bounding box`, and inverse (~). Clipping is very similar to intersection, except that clipping does not close any open surfaces (for example, a hemisphere appears to be hollow), and it is faster than intersections. Use bounding boxes to speed up ray tracing calculations, since they define the limits of an object or groups of objects. You should not use them as a clipping operation. The ray tracer simply tests the limits of the bounding box first; then, if necessary, it tests the objects inside it. This feature can save considerable time when you are ray tracing a large number of objects which are grouped together. Finally, use the inverse operation to define which part of an object (inside or outside) is important. If an object is located inside of another, applying an inverse to it causes it to be outside, and vice versa.

In the next chapter, you take a look at some additional features which can help you speed up the development process of images. In addition, you see how you can use mathematical expressions to increase Polyray's versatility.



Additional Features

So far, you have been able to create some interesting images in Polyray with only a fraction of its capabilities. You created high-quality images, using all of the available surface features: reflection, refraction, and highlights, to name a few. Occasionally, you might want to turn on or off some of those features to create a certain effect on your image, or to speed up the process. To turn on or off such features as reflection, refraction, or shadowing locally, you would have to go to each object and select them individually. In this chapter, you learn how to control some of those features globally, applying settings to a whole scene. You also learn ways of controlling the quality of the image generated, so that in the development phase, when you don't have to see high-quality images, you can sacrifice quality in order to get a faster preview. Finally, you take a look at Polyray's powerful math support.



Shading Flags

In Table 2.1 (which also is located on your tear-out card in the back of the book), one of the command-line parameters accepted by Polyray is -q. This parameter controls the quality of the shading. By reducing the amount of shading features, you can increase the speed of the ray tracer and have a quick turn-around of the final image, and do some interesting special effects.

Table 6.1 shows the options controlled by the -q parameter.

Table 6.1. Shading options controlled with command-line parameter -q.

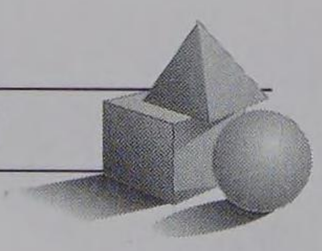
Value	Option
1	shadows
2	reflection
4	refraction
8	highlights (both sides of a surface)
16	check if object casts shadows
32	check primary light rays (rays that go from observer to object)

You can turn on or off any option by including or excluding it when defining the parameter. For example, assume you have an image that contains many reflective objects. Light bounces everywhere, causing the ray tracer to dedicate more time to checking the reflections. In order to speed up the image, you can turn off that option:

```
  1    shadows generation
  4    refraction
  8    highlights both sides of a surface
 16    shadows checking (determines if object casts shadows)
+32    primary rays (rays from the observer to the object)
 61    parameter value
```

If DEMO.PI is the scene file, the command-line parameter is:

```
POLYRAY DEMO.PI -q 61 [press the Enter key]
```



As you can see, the only option missing is the second one (reflection). You can change any number of options this way.

If you wish to control those same options on specific objects, Polyray has defined the `shading_flag` keyword, which also uses the -q parameter values.

Polyray also has defined some handy keywords that represent the values of the different options. These definitions are available in Listing 6.1, the TEXTURES.INC file: .

Listing 6.1. Defined shading flag keywords (taken from TEXTURES.INC).

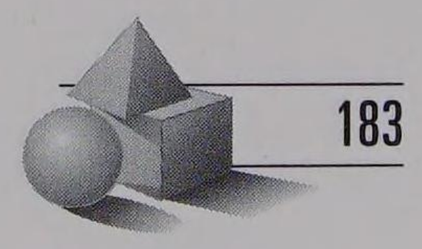
```
// Names of the various shading flags
define shadow_check          1
define reflect_check         2
define transmit_check        4
define two_sided_surface     8
define cast_shadows          16
define primary_rays          32

// Default shading flags during ray tracing—everything gets checked.
define all_shading_flags                  shadow_check
                                        + reflect_check
                                        + transmit_check
                                        + cast_shadows
                                        + two_sided_surface
                                        + primary_rays

// Shading flags for surfaces that shouldn't have their backsides illuminated
define one_sided_surface   all_shading_flags - two_sided_surface
```

Since Polyray understands mathematical expressions, the keyword `all_shading_flags` is defined by adding all of the possible options (each option represents a value).

If you use this method of defining shading flags for each object in CSG operations, you must apply the flags to each object in the definition.

Rendering Method

Ray tracing produces the highest quality output. Polyray supports two other rendering methods: scan conversion and wireframe. You can select which rendering method by using the -r command (see Table 2.1). Both methods are described here briefly.



Scan Conversion

Scan conversion handles all objects as a collection of polygons. You can improve the quality of the surfaces by increasing the number of polygons used to create the object. You can improve the smoothness of the surface by using two keywords: `u_steps` and `v_steps`. These keywords define the number of subdivisions to be performed on a given object. The `u_steps` keyword refers to the dimension you measure from left to right. The `v_steps` keyword refers to the dimension you measure from top to bottom. Note the following example:

```
object {
     sphere <0, 0, 0>, 2
     shiny_red
     u_steps 10
     v_steps 30
}
```

The sphere in this scene is divided from the north pole to the south pole into 30 segments, while the equator (from left to right) will be subdivided 10 times. Increasing the number of subdivisions increases the smoothness of the surface.

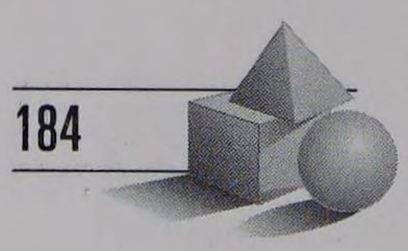
Scan conversion can be faster than ray tracing depending on the amount of `u_steps` and `v_steps`, but there is a catch—it requires substantial amounts of memory. You may find that with your current computer setup, you may be able to ray-trace scenes, but not scan-convert them due to the memory requirements.

Finally, when scan-converting a scene, use the command-line parameter -r1. Use the -q parameter also to explicitly turn on or off shading options, since scan conversion by default does not support shadows, reflectivity, or transparency.

Wireframe

Wireframe representation is a variation of scan conversion. Only the edges of the polygons are visible, giving the appearance that the objects are made with wires. One of the benefits of wireframe representation is that it generates the image the fastest. On the other hand, wireframe rendering does not support Boolean operations.

To select wireframe rendering, use the -r2 command-line option to select wireframe representation and -W to pause the screen. Figure 6.1 displays the scene file INVERSE.PI in wireframe representation. Note that none of the Boolean operations are shown in the image.



Mathematical Expressions

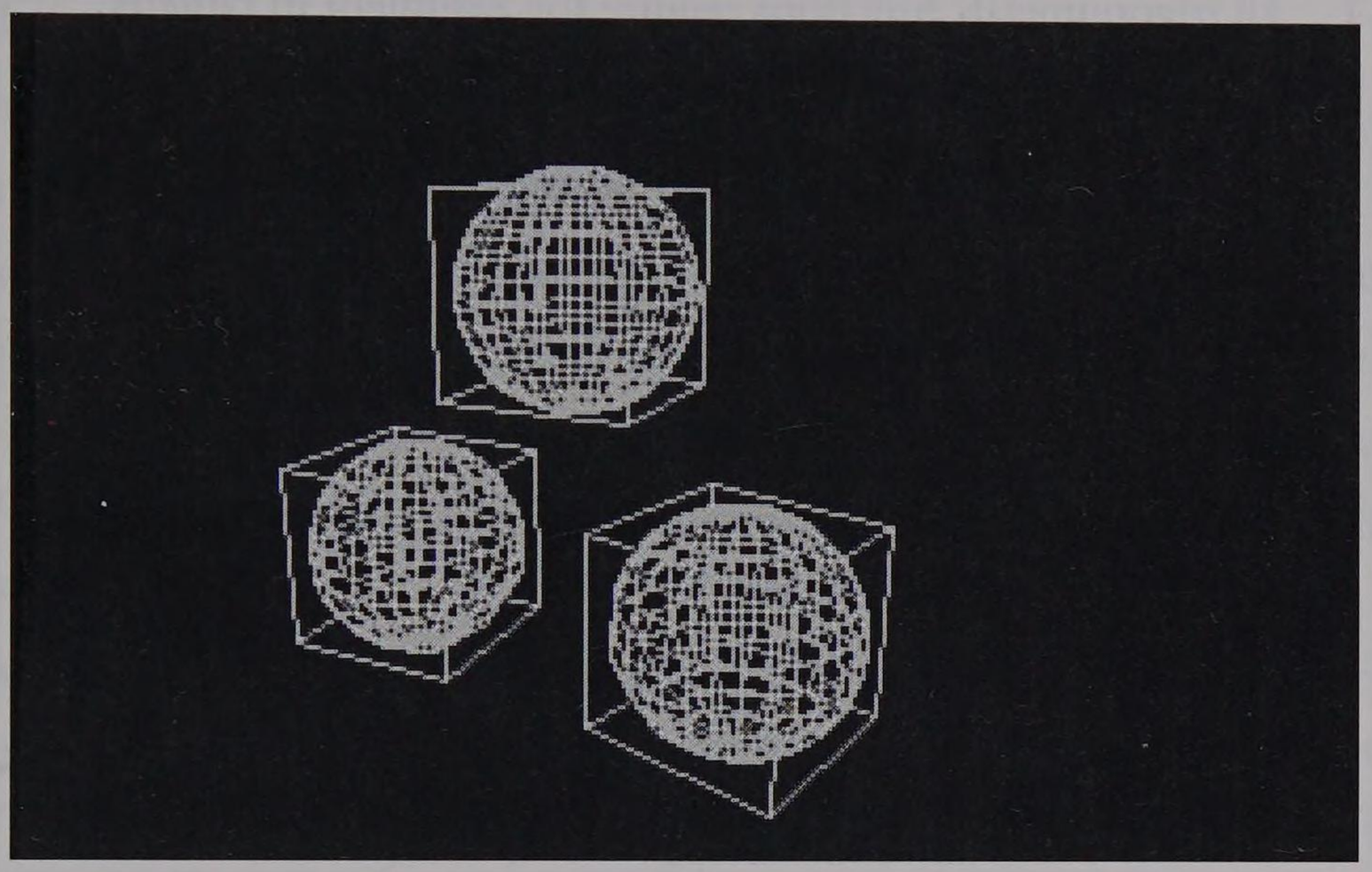


Figure 6.1
Wireframe representation of scene INVERSE.PI.

One of Polyray's most powerful features is the capability to understand mathematical expressions. There are four valid types of expressions: floating point, vectors, arrays, and conditional.

Floating Point Expressions

Floating point expressions, or "fexper," can consist of integer numbers (1, 5, –3) real numbers (0.05, 25.93, 100.04) alone or combined with functions that return real or integer numbers. Table 6.2 presents a list of available functions in Polyray that return a number. The available mathematical relations are:

(fexper)	Parenthesized expression
fexper 1^ fexper 2	fexper1 to the power of fexper2
fexper * fexper	Multiplication
fexper / fexper	Division
fexper + fexper	Addition
fexper – fexper	Subtraction
–fexper	Minus sign

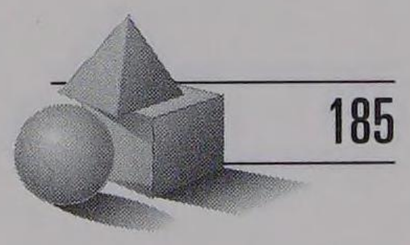
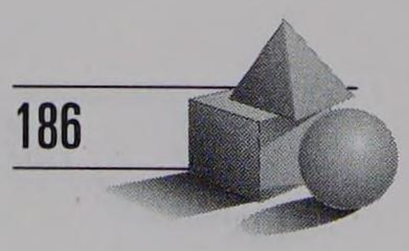


Table 6.2. Functions that return integer and floating point numbers.

Notes:

1. All trigonometric functions require the argument in radians
2. fexper is a floating point expression
3. vexper is a vector expression, for example: <23.6, 2*10.3, 0.0>

acos(fexper)	Arccosine
asin(fexper)	Arcsin
atan(fexper)	Arctangent
ceil(fexper)	Ceiling function, finds smallest integer not less than fexper
cos(fexper)	Cosine
cosh(fexper)	Hyperbolic cosine
degrees(fexper)	Converts radians to degrees
exp(fexper)	Standard exponential function
fabs(fexper)	Absolute value
floor(fexper)	Floor function, finds largest integer not greater than fexper
fmod(fexper, fexper)	Modulus function for floating point values
ln(fexper)	Natural logarithm
log(fexper)	Logarithm base 10
noise(vexper,fexper)	Correlated noise function based on a location vector <x,y,z>. fexper is the number of times the output will be fed back as a new input to generate a new pseudo-random number between 0 and 1.
pow(fexper1,fexper2)	Exponentiation (same as fexper1 ^ fexper2)
radians(fexper)	Converts degrees to radians



sawtooth(fexper)	Sawtooth function, range between 0 and 1
sin(fexper)	Sine
sinh(fexper)	Hyperbolic sine
sqrt(fexper)	Square root
tan(fexper)	Tangent
tanh(fexper)	Hyperbolic tangent
visible(vexper,vexper)	Returns 1 if second point is visible from the first
vexper[i]	Extract component i from a vector or an array
vexper . vexper	Dot product of two vectors
|fexper|	Absolute value (same as fabs)
|vexper|	Length or magnitude of a vector

Floating point expressions can be used anywhere a real number is required. Assume you want to create a sphere with the center located at the origin <0, 0, 0> and the surface touching the point <1, 1, 3>. To find the radius, you can calculate the distance between <0, 0, 0> and the point <1, 1, 3>:

vector from origin to point = point – origin

= < 1–0, 1–0, 3–0 >

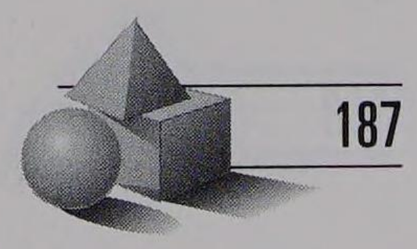
= <1, 1, 3>

the distance or magnitude of the vector is

$$\sqrt{x^2+y^2+z^2} \text{ or } \sqrt{1^2+1^2+3^2}$$

The magnitude of a vector can be written in Polyray as |<1, 1, 3>|. First, transfer these calculations to Polyray:

```
define origin < 0, 0, 0>
define m <1, 1, 3>
define radius ¦ m - origin ¦
object {
     sphere origin, radius
     shiny_green
}
```



In this case, you didn't need a calculator to find out any numbers; Polyray takes care of the calculations. By changing the location vectors origin and m, you can redefine new spheres.

Try another example. Assume you are interested in creating a group of spheres that will model a sine function. The sine function will be used as the y value:

```
define p1 <0,   sin(0), 0>
define p2 <5,   sin(radians(5)), 0>
// radians(5) = convert 5 degrees to radians
define p3 <10, sin(radians(10)), 0>
define p4 <15, sin(radians(15)), 0>
define p5 <20, sin(radians(20)), 0>
define p6 <25, sin(radians(25)), 0>
...ect...
object {
     sphere p1, 0.2
      ...ect...
}
object {
     sphere p2, 0.2
      ...ect...
}
object {
     sphere p3, 0.2
      ...ect...
}
.... ect ...
```

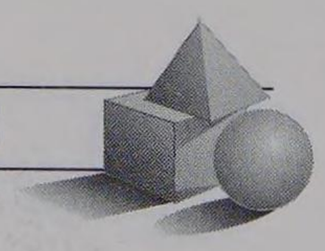
This previous example can be accomplished using arrays (presented later in this chapter).

Vectors

As you can see in the previous example, floating point expressions can be used inside vectors. You are already familiar with various vectors such as colors <R, G, B>, and the ones used in the viewpoint definition:

```
viewpoint {
     from <x, y, z>
     at <x, y, z>
     up <x, y, z>
     ...ect...
}
```

The keywords `from`, `at`, and `up` all use vectors to define locations and directions. Each of the values of a vector is called a *component* and can consist of any floating point expression.



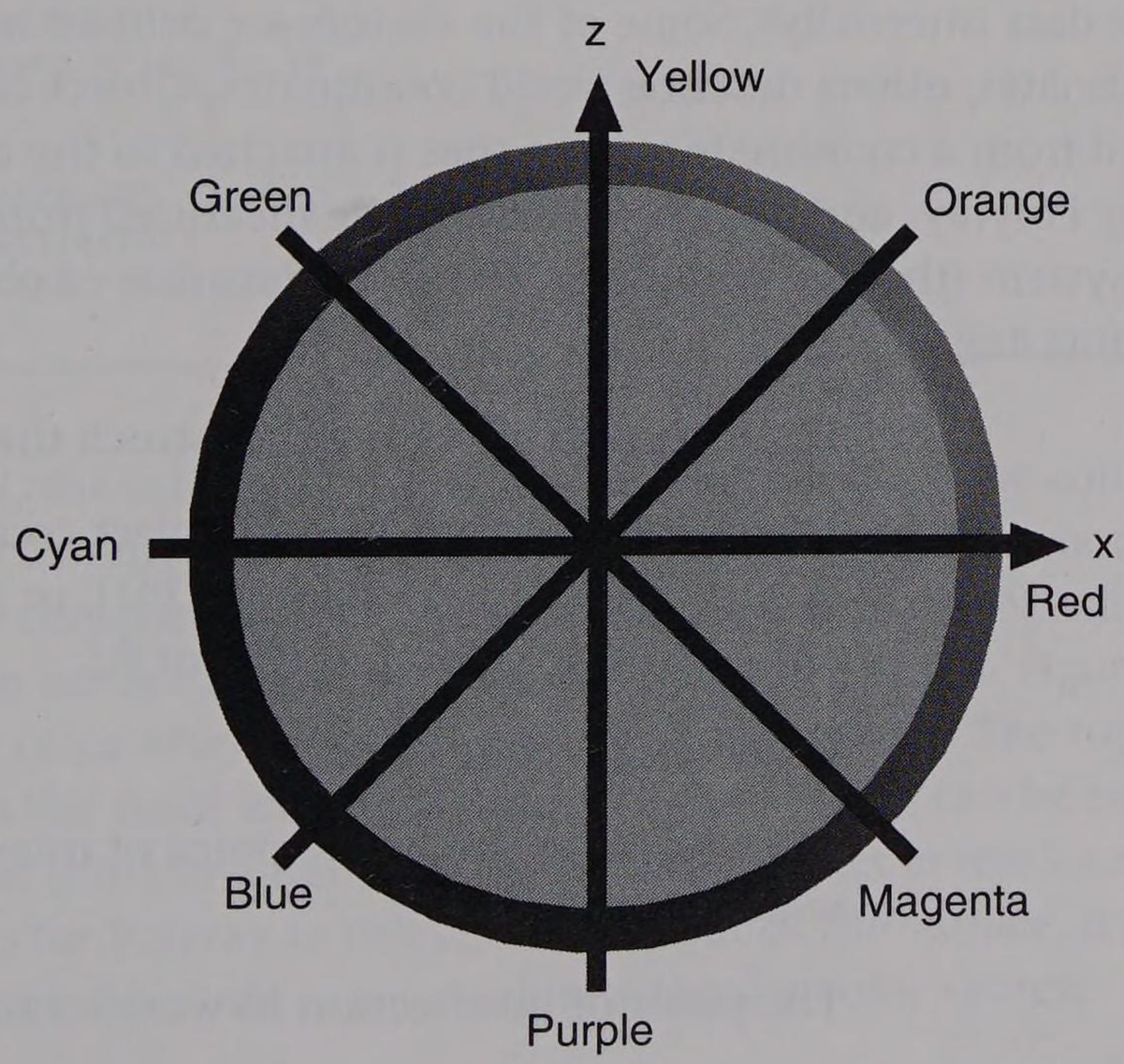
The operations you can perform between vectors and floating point expressions are

vexper + vexper	Addition
vexper – vexper	Subtraction
vexper * vexper	Cross product
vexper * fexper	Scaling of a vector by a scalar
fexper * vexper	Scaling of a vector by a scalar
vexper / fexper	Inverse scaling of a vector by a scalar

Table 6.3 presents a list of functions that return vectors.

Table 6.3. Functions that return vectors.

brownian(vexper)	Makes a random displacement of a vector
color_wheel (x, y, z)	RGB color wheel using x and z, the color returned is based on <x, z> using the chart below. Intermediate colors are generated by interpolation.



continues

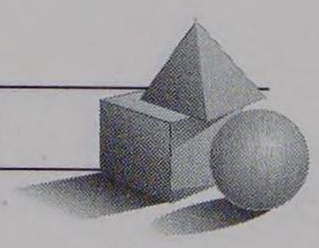
Table 6.3. Continued.

rotate(vexper,vexper)	Rotate the point given in the first argument by the angles given in the second argument. (Angles in degrees.)
rotate(vexper,vexper,fexper)	Rotate the point in the first argument around the axis given in the second argument, by the angle in the third argument. (Angle in degrees.)
reflect(vexper,vexper)	Reflect the first vector about the second vector (second vector is rotated about the first by 180 degrees).
dnoise(vexper,fexper)	Returns a pseudo-random vector where each component is between 0 and 1. The value fexper determines the number of times the output will be fed back to generate a new random vector.

In addition to the functions presented in Table 6.3, Polyray has a set of vectors that are defined at run time (not in the scene file but as Polyray analyzes the data internally). Some of the vectors are defined in terms of object coordinates; others describe world coordinates. Object coordinates are measured from a coordinate system that is attached to the object internally by Polyray, and world coordinates are measured from the global coordinate system (the one you use to define the location of objects). The defined vectors are:

I	The direction of the ray that struck the object.
P	The point of intersection in object coordinates. (You can also use x = P[0], y = P[1], or z = P[2] to represent single components of P.)
x, y, z	Single components of P.
N	The normal vector at the point of intersection in world coordinates.
W	The point of intersection in world coordinates.

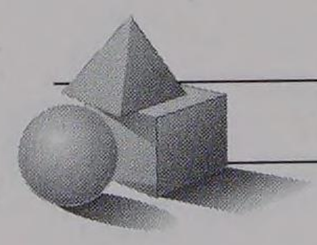
Listings 6.2 and 6.3 present an example of using the defined vectors in Polyray.



Listing 6.2. Using the color wheel function (WHEEL1.PI).

```
//WHEEL1.PI
define WheelTexture
texture {
   special surface {
      color color_wheel(x, y, z)
      normal N
      ambient 0.2
      diffuse 0.8
      specular white, 0.2
      microfacet Reitz 10
      }
}
include "colors.inc"
include "texture.inc"
viewpoint {
 from      <1, 1.3, -7.0 >
 at        <0.0, -1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
directional_light < 10 ,  10 , -10  >
// LIGHT_SOURCE
directional_light <-10 ,  10 , -10  >
background grey
    object {
        box <-1, -1, -1>, <1, 1, 1>
        WheelTexture
        rotate <-10, 0, 0>
    }
   object {
        sphere <0, -3, 0>, 1.2
        WheelTexture
        rotate <-10, 180, 0>
    }
```

In Listing 6.2, the `color_wheel` function I use, to calculate the color value based on the current point intersected by the light ray. The coordinates x, y, and z are the components of P. In addition, the normal is N, which is the normal of the surface at the point where light intersects it. Figure 6.2 shows the resulting color wheel applied to a cube and a sphere. The top of the cube displays the color wheel variation. The procedure can be summarized by saying that initially a light ray intersects an object at the location P (or x, y, z). In order for Polyray to calculate the color of the surface, it substitutes the x, y, and z values into `color_wheel`, returning a color vector.



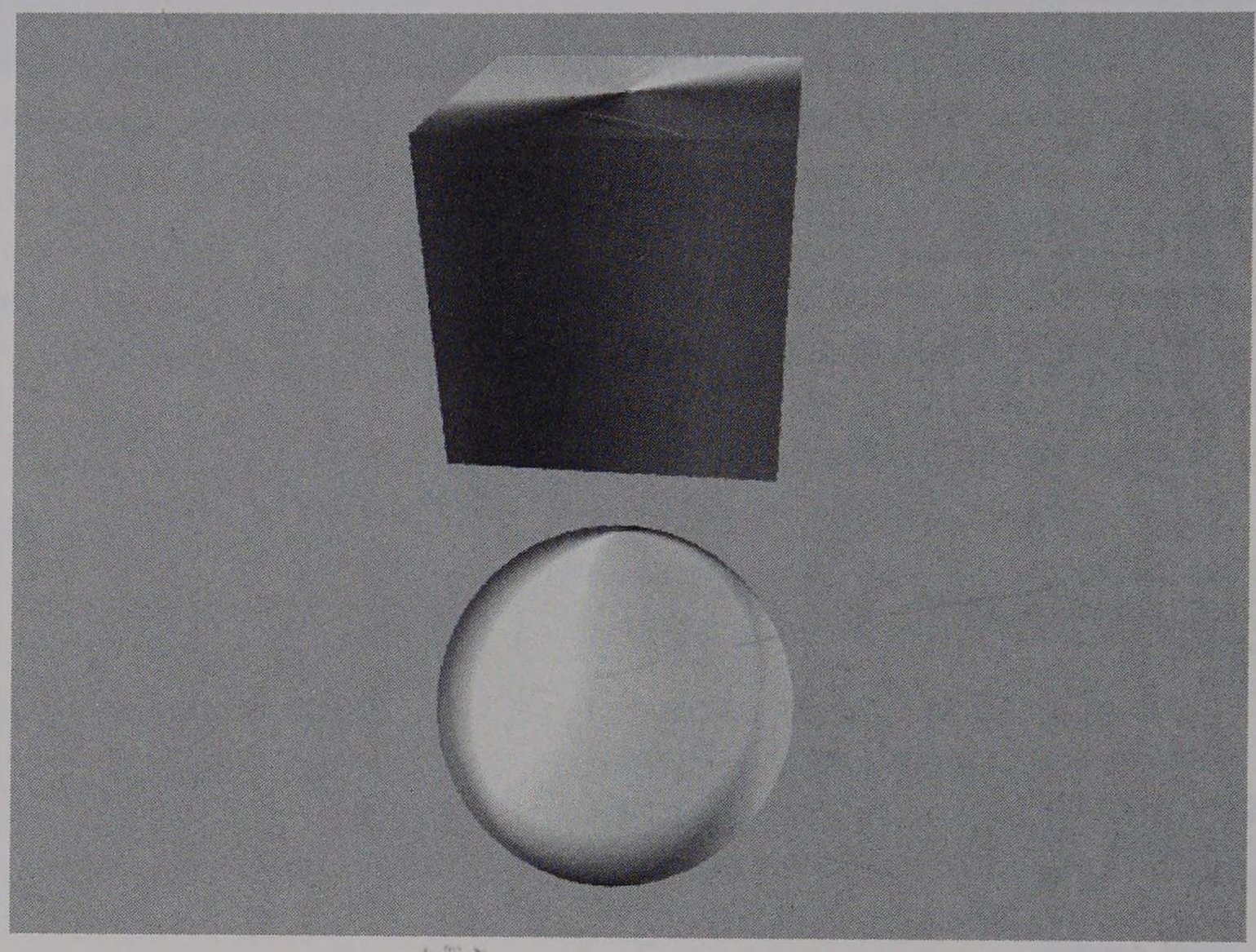


Figure 6.2

Using the color wheel function (WHEEL1.PI).

In order to cause turbulence, there are two approaches: create a disturbance in the color function or deform the surface by modifying the normal. Listing 6.3 uses both approaches. Figure 6.3 shows the result.

Listing 6.3. Creating turbulence in Listing 6.2 (WHEEL2.PI).

```
//WHEEL2.PI
define WheelTexture
texture {
   special surface {
      color color_wheel(x, y, z) + 2*(dnoise(3*P,3) - <0.5, 0.5, 0.5>)
      normal N + 10*(dnoise(P,3) - <0.5, 0.5, 0.5>)
      ambient 0.2
      diffuse 0.8
      specular white, 0.2
      microfacet Reitz 10
      }
}
include "colors.inc"
include "texture.inc"
viewpoint {
 from      <1, 1.3, -7.0 >
 at        <0.0, -1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
```

```
directional_light < 10 ,  10 , -10  >
// LIGHT_SOURCE
directional_light <-10 ,  10 , -10  >
background grey
    object {
        box <-1, -1, -1>, <1, 1, 1>
        WheelTexture
        rotate<-10, 0, 0>
    }
   object {
        sphere <0, -3, 0>, 1.2
        WheelTexture
        rotate<-10, 180, 0>
    }
```

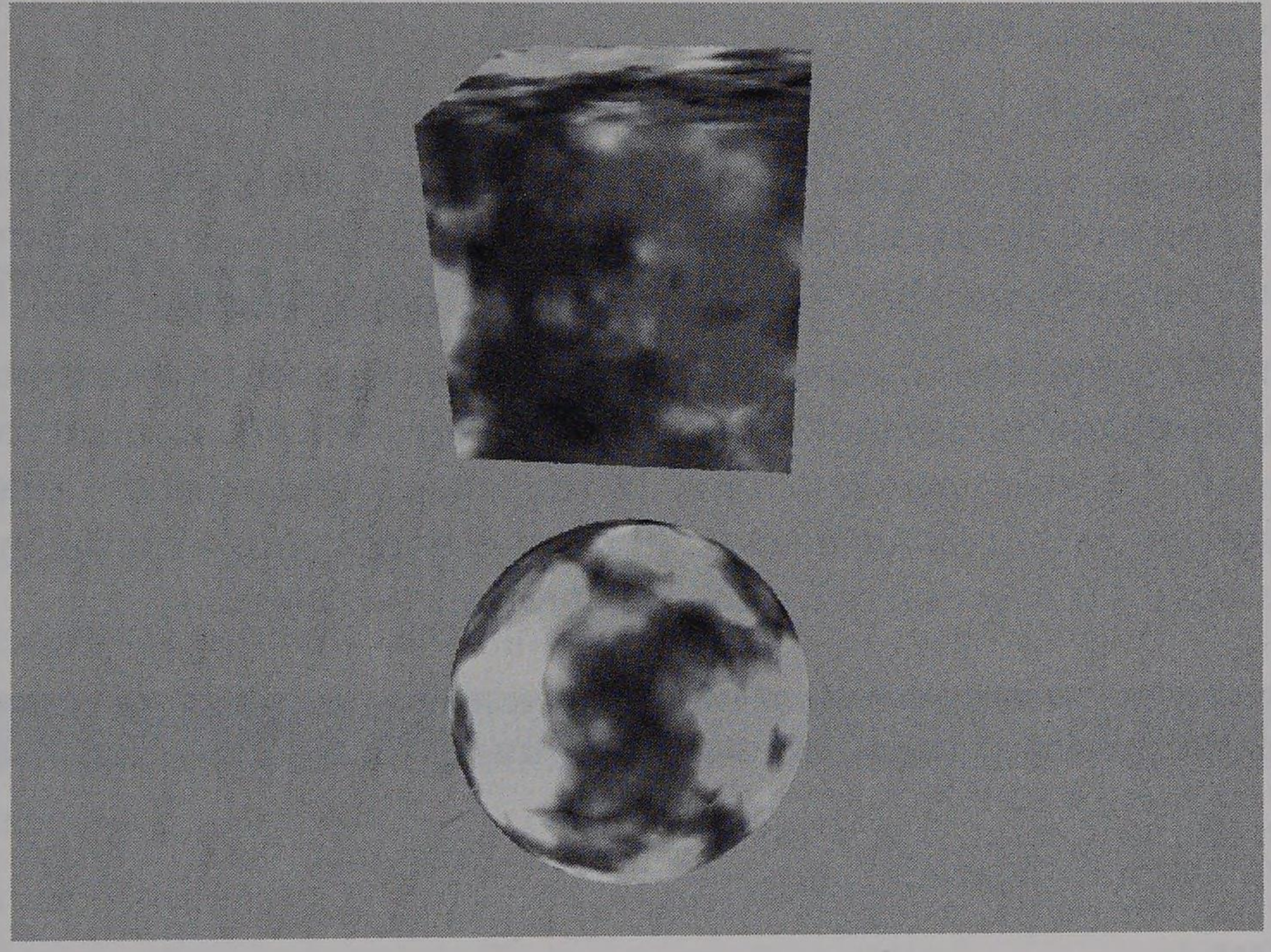


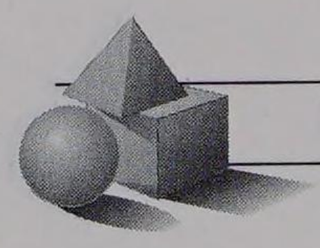
Figure 6.3

Creating turbulence by modifying surface colors and normals (WHEEL2.PI).

In Listing 6.3 the color is affected by adding a noise function:

```
color color_wheel(x, y, z) + 2*(dnoise(3*P,3) - <0.5, 0.5, 0.5>)
```

The `dnoise(3*P,3)` function takes the ray intersection point and scales it by a factor of 3. The second 3 is the amount of times that the random-generated vector is fed back to the `dnoise` function to increase its randomness (this number is usually referred to as *octaves*). Since the resulting vector will have components in the range of 0 to 1, the vector `<0.5, 0.5, 0.5>` is used to reduce the total effect. Finally, the resulting vector is scaled by 2 and added to the color vector generated by `color_wheel(x,y,z)`.



The normal of a surface can be modified by adding a disturbance to the original normal (N), the same way we did with the surface color:

```
normal N + 10*(dnoise(P,3) - <0.5, 0.5, 0.5>)
```

Again, a disturbance is added to the normal N. This will have the effect of changing the orientation of the surface; for example, if a flat surface is used, the changes in the normal vectors will make the surface appear bent.

Arrays

Polyray can group sets of expressions (fexper, vexper) into lists called arrays. The syntax for an array is

```
define ArrayName [ exper#1, exper#2, exper#3,...,exper#N]
```

where `exper#1, exper#2, exper#3, ..., exper#N` can be floating point expressions or vectors. Arrays can have more than one dimension; in other words, an array can have elements which are arrays themselves.

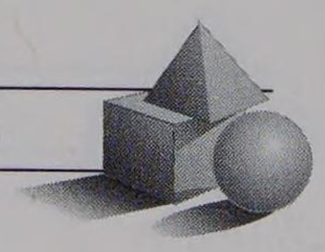
The next example uses three arrays to hold red, green, and blue color values and a fourth array to save vector positions. To select specific elements of any array, the array name is followed by the index (number) of the array element enclosed in square brackets (the first element starts with the index 0). Listing 6.4 is a section of a scene that presents this method.

Listing 6.4. Using arrays (section of a scene file).

```
define R [0, 0.5, 0.7, 0.9]
define G [0.1, 0.15, 0.6, 0.65]
define B [0.05, 0.4, 0.45, 0.77]
define  POS [<0, 0, 0>, <1, 0, 0>, <0, 1, 0>, <0, 0, 1>]
object {
     sphere POS[0],2
     texture{ surface {color <R[2], G[2], B[3]> }}
}
object {
     sphere POS[1],2
     texture{ surface {color <R[1], G[2], B[0]> }}
}
```

The first sphere is located at `<0, 0, 0>` and has the surface color `<0.7, 0.6, 0.77>`. The second sphere is located at `<1, 0, 0>` and the color is `<0.5, 0.6, 0.05>`.

If you recall the example where we were positioning spheres to model a sine function, we can replace the code using arrays:



```
define Loc  [  <0,   sin(0), 0>,
          <5,   sin(radians(5)), 0> ,
          <10, sin(radians(10)), 0>,
          <15, sin(radians(15)), 0>,
          <20, sin(radians(20)), 0>,
          <25, sin(radians(25)), 0> ]
object {
     sphere Loc[0], 0.2
      ...ect...
}
object {
     sphere Loc[1], 0.2
      ...ect...
}
object {
     sphere Loc[2], 0.2
      ...ect...
}
```

In Chapter 8, you'll take a look at how arrays can be used in animation.

Conditional Expressions

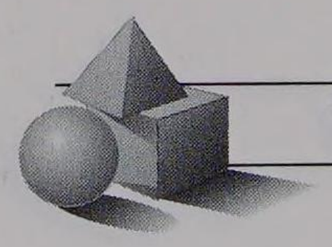
Conditional expressions have two values depending on the status of a certain expression. If the expression is True, one value is given; if it's False, the other is returned. The syntax for a conditional expression is

```
(cexper ? true_value : false_value)
```

where cexper has one of the following forms:

Conditional Expression	Returns True If
!cexper	NOT(cexper)
cexper && cexper	cexper AND cexper
cexper || cexper	cexper OR cexper
fexper < fexper	fexper LESS THAN fexper
fexper <= fexper	fexper LESS THAN OR EQUAL TO fexper
fexper > fexper	fexper GREATER THAN fexper
fexper >= fexper	fexper GREATER THAN OR EQUAL TO fexper
fexper == fexper	fexper IS IDENTICAL TO fexper
vexper == vexper	vexper IS IDENTICAL TO vexper

and `true_value` or `false_value` can be a floating point expression or a vector expression.



Listing 6.5 presents two examples of conditional expressions. The first expression defined as "K"

```
define K (x >= -1 && x <= 1 && y >= -1 && y <= 1 ?  white: red)
```

determines the color of the surface. The condition can be described as "IF x is between the values of –1 and 1 AND y is also between the values of –1 and 1, THEN the color of the surface will be white, ELSE the color will be red."

The expression defined as "T"

```
define T (x >= -1 && x <= 1 && y >= -1 && y <= 1 ?     1 : 0)
```

can be described as "IF x is between the values of –1 and 1 AND y is also between –1 and 1 THEN return a value of 1, ELSE a zero is returned."

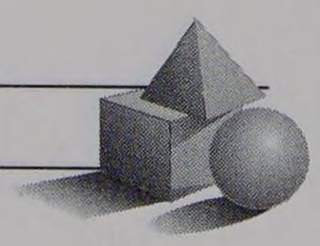
Recall that x, y, and z are the components of the point P defined by Polyray, and they refer to the point where the current light ray intersects the surface of the object. The coordinate system is "local" (attached to the object).

Finally, the texture definition for the outer sphere,

```
texture {
  special surface {
      color K
      ambient 0.3
      diffuse 0.1
      specular 0.1
      reflection white, 0.3
      transmission T ,1
  }
}
```

uses both K and T. Observe that T is used to determine if a given area of the surface is transparent or not. Another important point is that in order to use the run-time variables I, N, P, x, y, z, and W inside of a texture, you must use the "special surface" texture definition and not "surface" (refer to Chapter 4, sections "Texture Surface Parameters" and "Functional"). Recall that at run time the "special surface" definition can evaluate expressions that make reference to the variables I, N, P, x, y, z, and W.

Now that you have seen the major components of the scene, here is the rationale behind it. Your goal is to create an object that has a "window" to let you see whatever is inside of it. You can accomplish this with Boolean operators, but we are looking for an alternate method. First, the window has two distinct features: it's transparent and colorless. Second, the window has limits, which we chose as x between the values of –1 and 1 and y between the values of –1 and 1. The z value is not important since the window will be located in the front (where light rays hit first).



To take care of the transparency requirement, the expression T changes to 1 (fully transparent) when the light ray hits the window, and to 0 when it is outside of it. The color is approached the same way—it is white when the light is inside the window and red when touching the rest of the sphere. To test the effectiveness of this approach, create a blue sphere inside the outer one. If the window is transparent, you'll be able to see it; if the window doesn't act as a filter, the inner sphere will be blue. Figure 6.4 displays the final image. A checkered plane was positioned on the back to see if a second window is created in the back of the sphere. (it isn't.)

There are better ways to simplify the K and T expressions. An alternate approach could be

```
define Status (x >= -1 && x <= 1 && y >= -1 && y <= 1 ?  1: 0)
define K (Status == 1? white: red)
define T  Status
```

Try experimenting with other methods.

Listing 6.5. Using conditional expressions (COND.PI).

```
//COND.PI
include "colors.inc"
include "texture.inc"
define Plane_YZ  object {
                  polygon 4, <0,-10000,10000>, <0,10000,10000>,
                             <0,10000,-10000>, <0,-10000,-10000>
                  }
define K (x >= -1 && x <= 1 && y >= -1 && y <= 1 ?  white: red)
define T (x >= -1 && x <= 1 && y >= -1 && y <= 1 ?     1 : 0)
viewpoint {
 from      <1, 1, -7.0 >
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
light < 5 ,  5 , -10  >
// LIGHT_SOURCE
light <-5 ,  5 , -10  >

object {
   sphere <0, 0, 0>, 2
   texture {
    special surface {
      color K
      ambient 0.3
      diffuse 0.1
      specular 0.1
      reflection white, 0.3
```

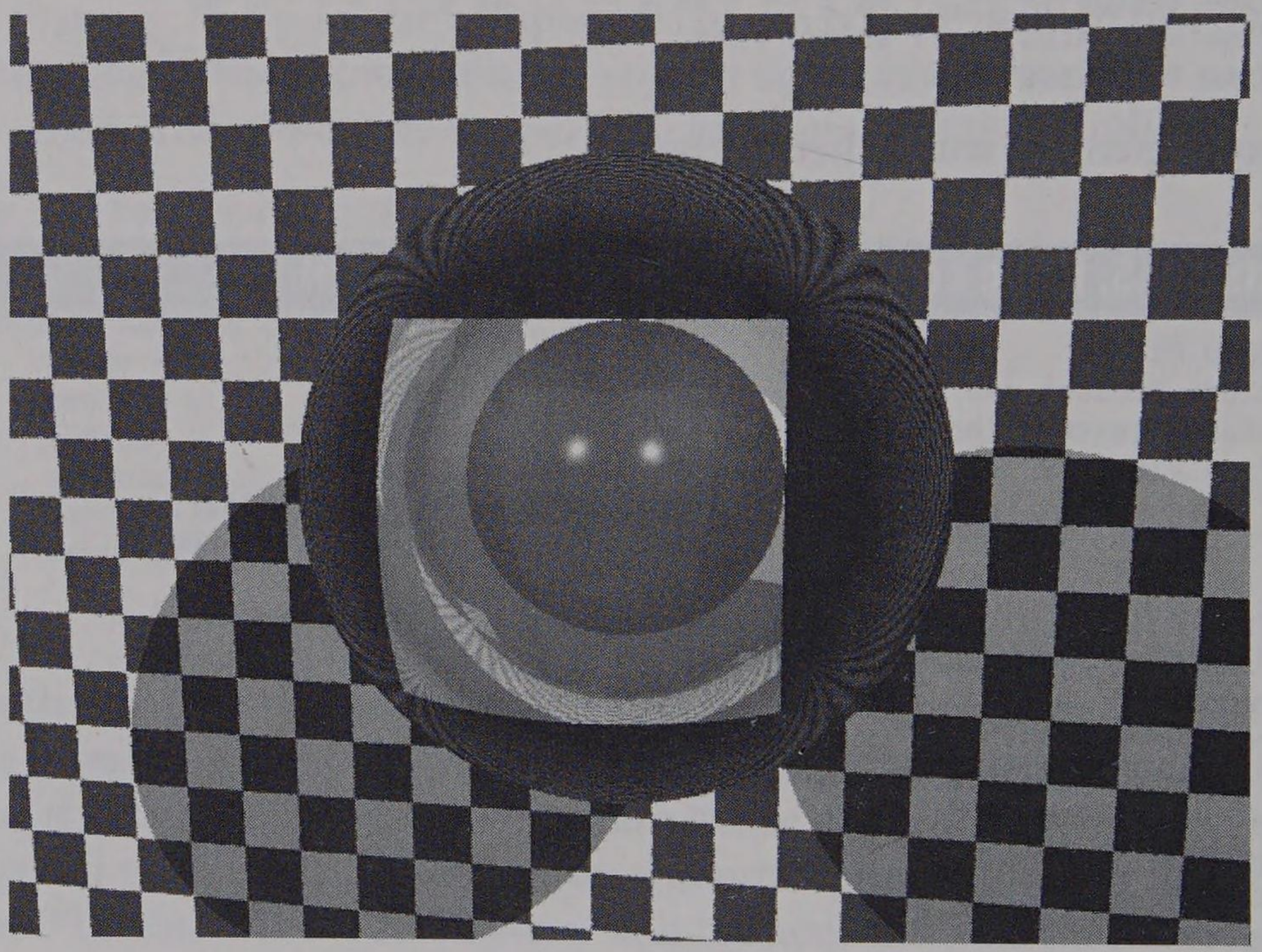
continues

Listing 6.5. Continued.

```
transmission T ,1
     }
   }
}
object {
   sphere <0, 0, 0> , 1
   shiny_blue
}
object {
  Plane_YZ
  rotate <0, 90, 0>
  translate <0, 0, 10>
  texture {checker matte_white, matte_black}
}
```

Figure 6.4

Creating a window on a sphere with conditional expressions (COND.PI).



Summary

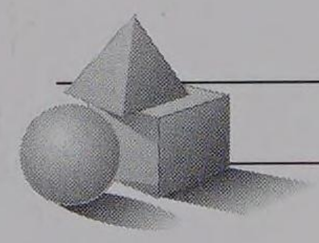
In this chapter, you saw different ways of speeding up the rendering process, either by reducing the quality of the image or by using alternate rendering methods. To reduce the quality of the image for faster viewing, you can disable some of Polyray's shading features using the -q command-line parameter or the keyword `shading_flag` inside of the scene file. Another way to speed up the image-creation process is to select scan conversion or wireframe representation. Both are fast methods, but scan conversion

requires a large amount of memory, and wireframe representation can only be used to preview the geometry. Wireframe representation does not support Boolean operations.

In addition, you took a detailed look at Polyray's powerful math support. Polyray can understand a wide variety of floating point and vector functions which you can use throughout the scene. Another interesting feature is the use of the run-time variables `I`, `N`, `P`, `x`, `y`, `z`, and `W`, which give information concerning the direction of the light ray, and the location where it intersected a given object. You can use these to modify the color and characteristics of the surface.

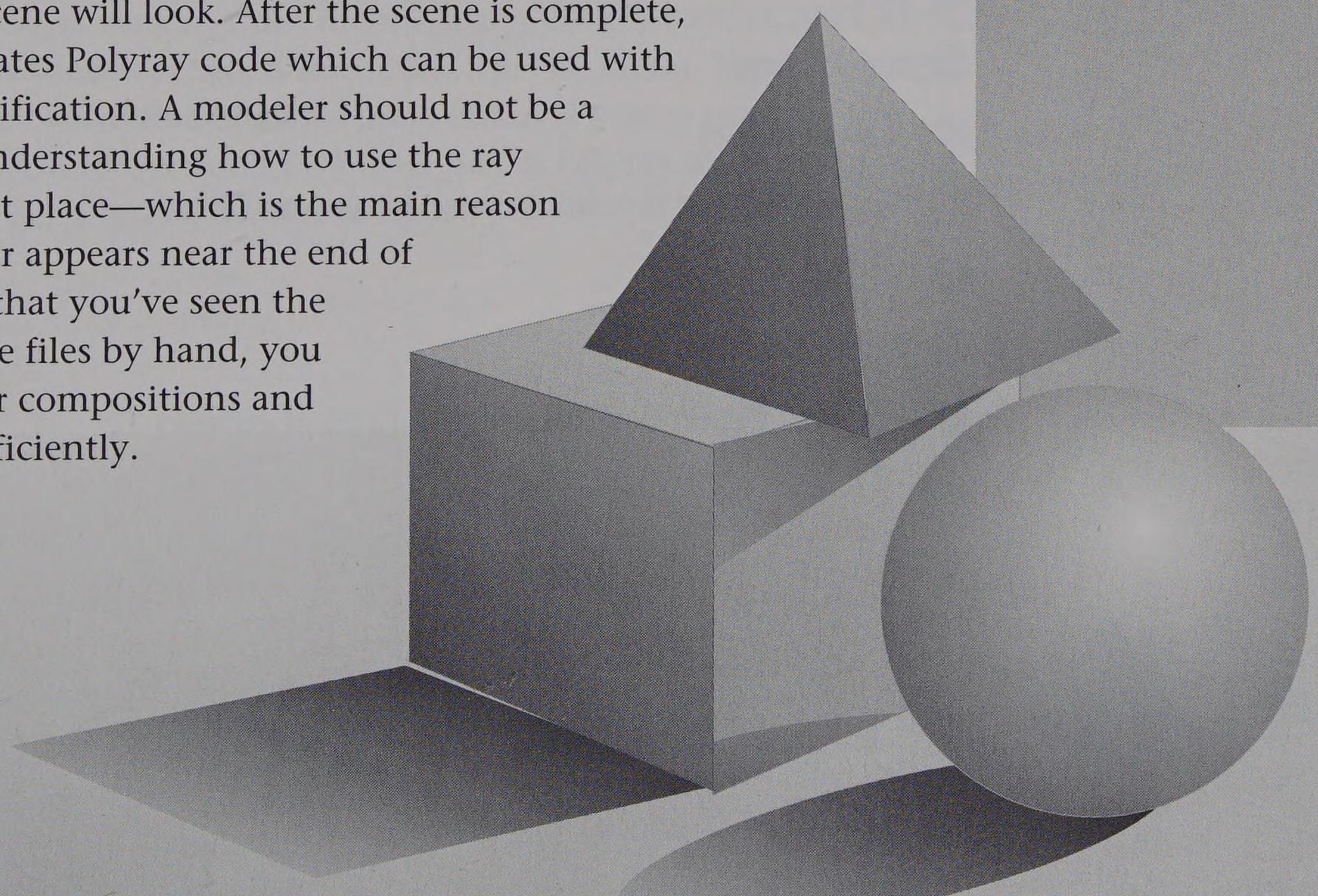
Arrays are collections of vectors or expressions you can organize in list form. You can select individual elements of an array by using the array name followed by the element index enclosed in square brackets . Finally, you studied conditional expressions, which are expressions that return two values or vectors depending on the relationship between two expressions.

In the next chapter, you'll be introduced to 3-D modeling as a tool for creating scenes visually. You'll have the chance to use POVCAD, a wireframe-based 3-D modeler that can help you create different scenes by positioning objects visually. You'll take a look at POVCAD's major features and have the opportunity to create scenes for the Polyray ray tracer.



Using a 3-D Modeler (POVCAD)

Included with this book is POVCAD, a wireframe-based modeler which will be used as a front-end for Polyray. POVCAD is a 3-D modeler you can use for prototyping your own scene files. By visually positioning the objects on-screen, you can get a better idea of how the final scene will look. After the scene is complete, POVCAD generates Polyray code which can be used with little or no modification. A modeler should not be a substitute for understanding how to use the ray tracer in the first place—which is the main reason why this chapter appears near the end of the book. Now that you've seen the creation of scene files by hand, you can create better compositions and modify them efficiently.

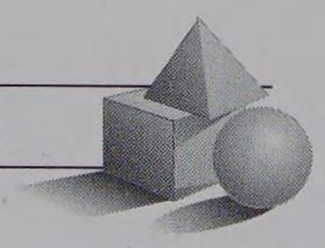


The chapter is a collection of tutorials. Each scene you'll see created will have the name TUTOR#.PI; # will be replaced with the corresponding tutorial number. It is beneficial to study how the modeler creates each scene file.

Introduction

I wrote POVCAD some time ago because of my own need to be able to visualize how objects were located in a scene file. Ray tracing the image every time you move an object puts stress on the user as well as the computer! Use a modeler as a tool to help you go beyond the worries of correct syntax, typing, and remembering the format of a function, and get on with developing a more interesting scene. Here is a list of objects you can create with POVCAD:

- plane
- cylinder
- cone
- sphere
- ellipsoid
- torus
- box
- height field
- Raw data files
- Sweep (solid of revolution, also known as Lathe)
- Extrude
- Bézier patches
- Polygon Decomposition (tessellation) of curves



- light sources
- viewpoint definition

This is by no means a complete list. Polyray has more features than those supported by POVCAD. As time progresses I'll be adding other features to POVCAD. You'll become familiar with POVCAD's features as you develop your own scene files.

Note to DOS and Windows users: POVCAD is available in both versions. Usually the Windows version is ahead of the DOS version in terms of features. This chapter presents the features included in the Windows version of POVCAD. Some of these features might not be in the corresponding DOS version, but they should be added in future releases.

Running POVCAD

The program is usually distributed with the name POVCAD.EXE or possibly POVCAD*?*.EXE where *?* is replaced by a number. I'm assuming you have installed the program following the instructions in the READ.ME file that comes with the disk.

To run the DOS version, simply make the directory where POVCAD resides the current directory and type

POVCAD [press the Enter key]

To run the Windows version from file manager, click the POVCAD.EXE file; to run it from the desktop, select **Run** from the **File** menu. Type the path and name of the executable file, and press Enter. If you're running the Windows version, after running POVCAD, select Polyray as your ray tracer and you should get a screen similar to figure 7.1.

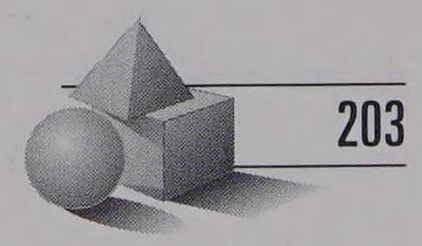


Figure 7.1

Initial POVCAD screen (Windows version).

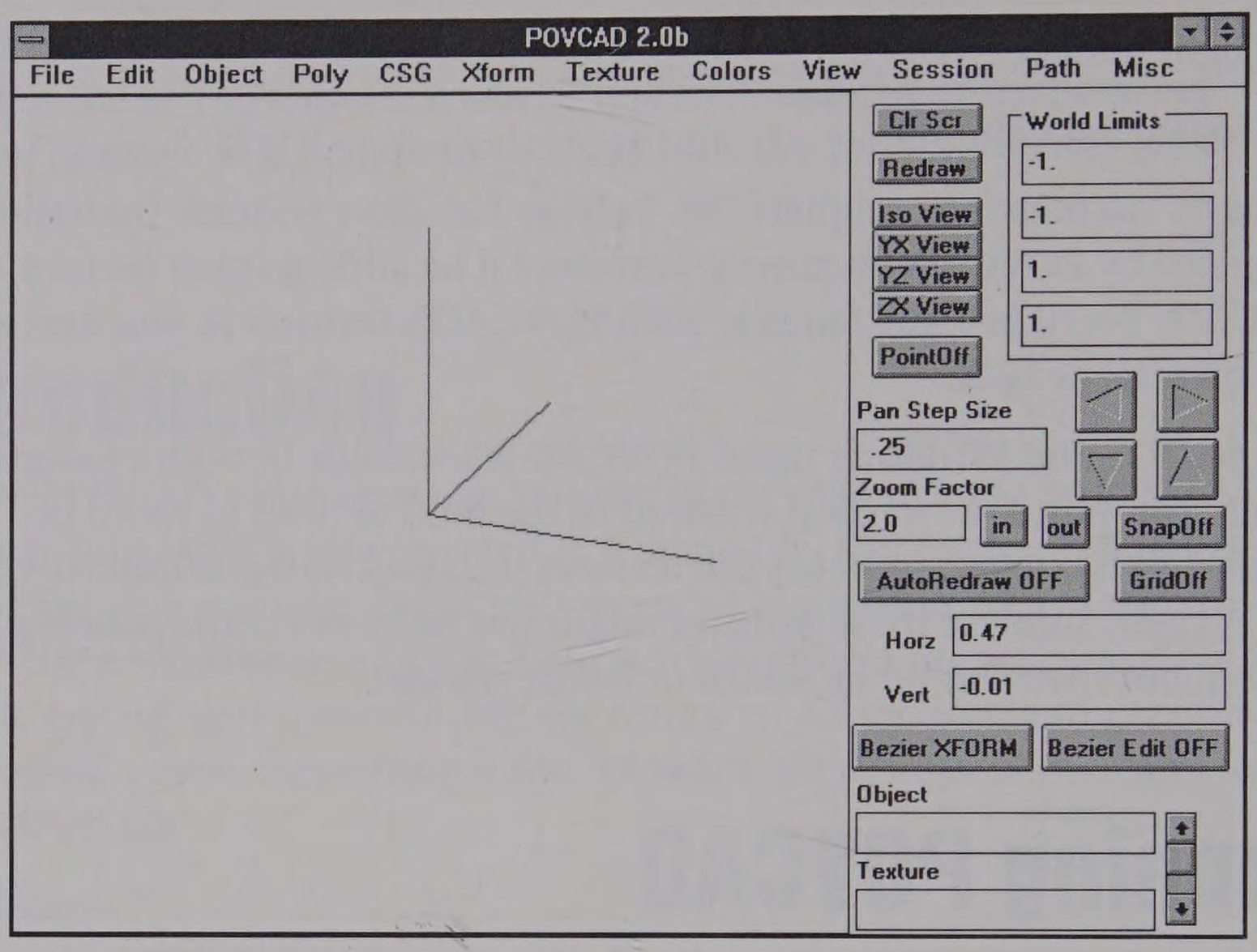


Figure 7.2 shows a section of figure 7.1 describing the name of the icons used for quick access to some of POVCAD's commands.

Figure 7.2

Icon section in POVCAD.

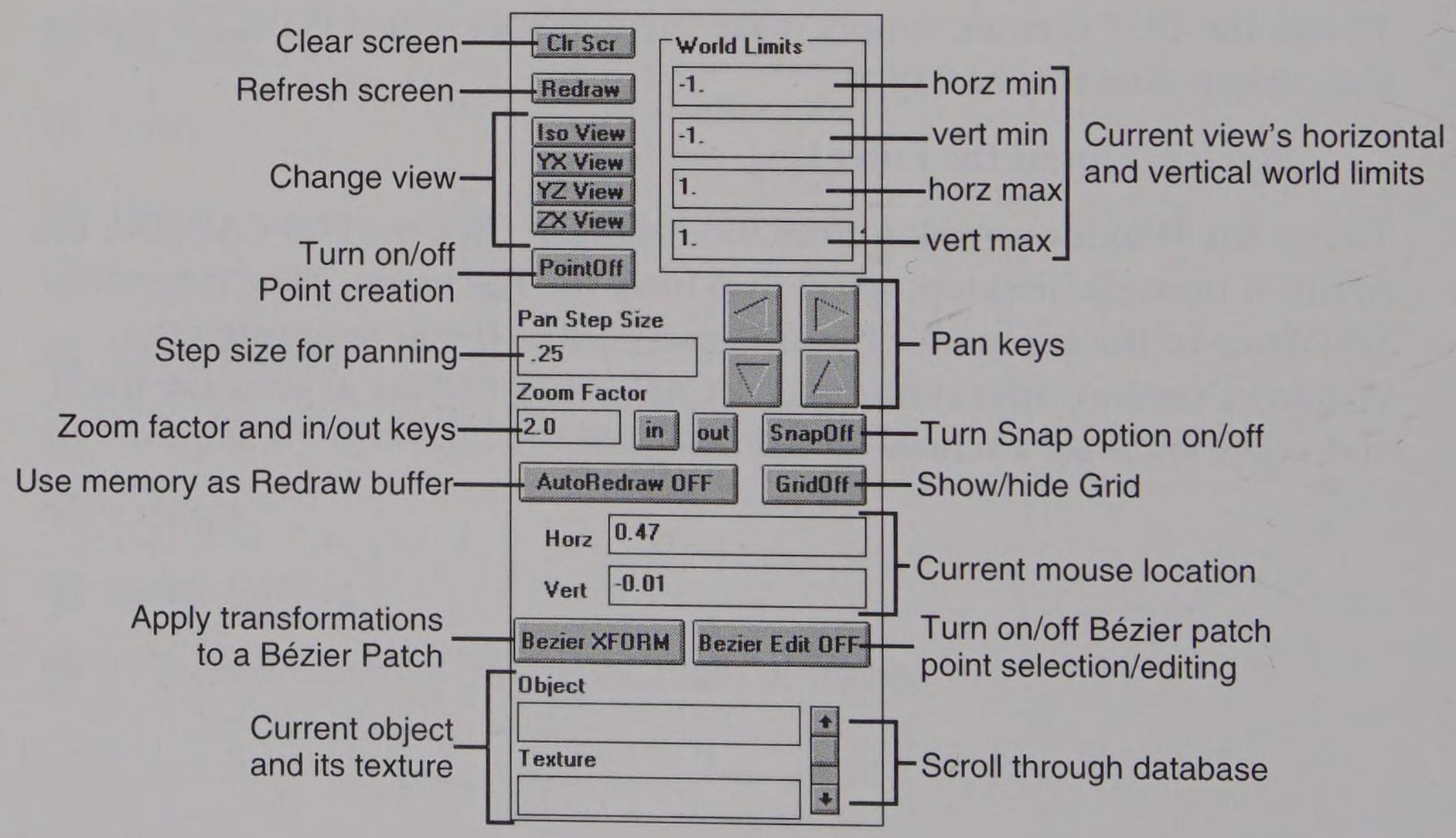


Table 7.1 describes each icon in more detail:

Table 7.1. The POVCAD icons.

Icon	Description
Clr Scr	Clears the drawing area, leaving only the 3-D axis.
Redraw	Redraws the complete scene and 3-D axis, if grid is ON it will be refreshed too.
Iso View	Scene is presented in a perspective point of view.
YX View	Scene is presented with y as the vertical axis and x as the horizontal axis.
YZ View	Scene is presented with y as the vertical axis and z as the horizontal axis.
ZX View	Scene is presented with z as the vertical axis and x as the horizontal axis.
World Limits	Displays the limits of the horizontal and vertical axis. The order of the numbers from top to bottom is: horizontal minimum vertical minimum horizontal maximum vertical maximum To change these numbers, from the View menu select the World command.
PointOff	Turn ON/OFF point creation routine. If status is ON, then any point clicked on the drawing area is saved and a curve is created.
Left, Right, Down, Up	Displaces the scene in the appropriate direction (Pan). The increment of this displacement can be modified by changing the number inside the Pan Step Size text box.
in, out	Zoomin and Zoomout keys. The zoom factor can be modified by changing the number inside the Zoom Factor text box.

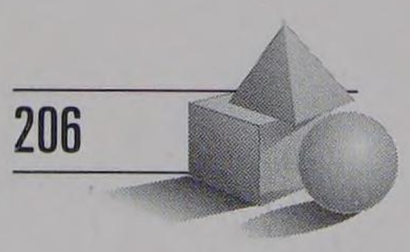
continues

Table 7.1. Continued.

Icon	Description
SnapOff	Turn snap funtion ON/OFF. If status is ON any point created will be moved to the nearest grid point.
AutoRedraw OFF	Selects between saving the drawing area to a buffer in memory (redraw is needed less) or continuous redraw. Enabling AutoRedraw (ON) will consume more memory. Normally is disabled.
GridOff	Show or hide grid.
Horz, Vert	Current mouse location.
Bézier XFORM	Apply transformations to a selected Bézier Patch.
Bézier Edit OFF	Enable/disable editing of current selected Bézier Patch.
Object	Displays current selected or created object.
Texture	Displays current selected or created object's texture. ScrollBar (located to the right of Object and Texture) scrolls through database of created objects. To view the complete list select the Session command.

The top menu selections are:

File	File operations.
Edit	Select objects or points to be deleted.
Object	Select from the different primitives (sphere, cone, sweep, and so on).
Poly	Create flat plates with holes.
CSG	Boolean operations menu.
XForm	Applies transformations to objects and curves.
Texture	Select from a list of predefined textures.
Colors	Select from a list of predefined colors.
View	Changes world limits, views (Iso, YX, and so on), grid spacing.



Icon	Description
Session	Displays a complete list of all the objects in the database.
Path	Saves points created with the mouse, applies curve smoothing and other data file operations.
Misc	Create light sources, define the viewpoint of the scene, select setting for Bézier patch displaying.

The best way to learn how to use POVCAD is by creating your first scene. In the next section, you create a few spheres and attach textures to them. In addition, you locate a few light sources and define the viewpoint. Finally, you see a Polyray scene file created and studied.

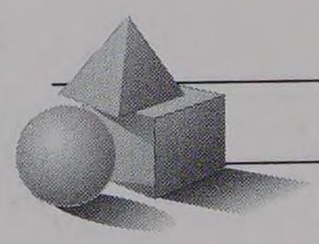
Creating a Scene File (Tutorial #1)

In this section you go through the process of creating a quick scene, which is basically the same for any scene you develop. These are the steps to create TUTOR1.PI:

1. From the Object menu select the Sphere command.
2. Enter the following values as they are requested (press Enter after each answer):

Radius	0.3
x	0.0
y	0.0
z	0.0

3. Click the YX button to change the view. Now y is the vertical axis and x is the horizontal axis.
4. Click the center of the sphere. This selects the object.
5. Move the mouse to the location Horz = 0.53 and Vert = –0.53 (check the Horz and Vert mouse position boxes).
6. When the Copy/Move window appears, select Copy with the right mouse button.



At this point, a copy of the original sphere should have been created and located at the position where you pressed the right mouse button. If you made a mistake, select Delete All from the File menu and start again at step 1.

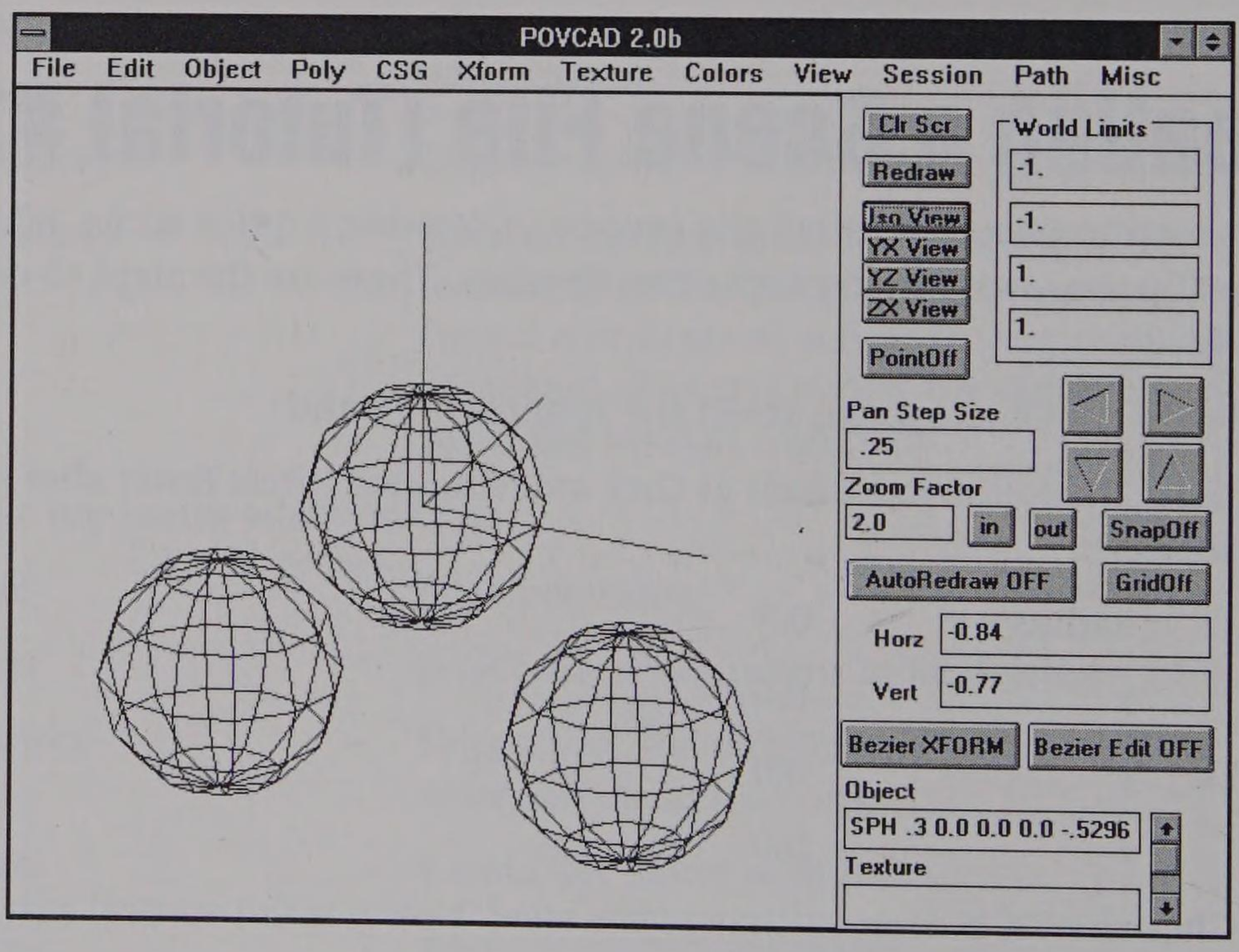
Continue with the following steps:

7. Select the sphere located at the origin by clicking the left mouse button at the center of the sphere.
8. Move the mouse to the location Horz = –0.53 and Vert = –0.53.
9. Press the right mouse button; when the Copy/Move window appears, select Copy.

Another copy of the sphere has been placed at the location where the right mouse button was pressed. Click the Iso View button to change the view. Figure 7.3 shows the image that should appear on your screen.

Figure 7.3

Three spheres created in POVCAD.



The next steps attach textures to the spheres. If you define a texture for the first sphere and copy it, all the copied spheres will have the same texture. This is why you perform these steps after creating all the objects.

1. Select the center sphere by clicking the left mouse button on the center of the object.

2. After clicking on the Texture command, click the down arrow to display a list of textures.
3. Select the reflective_grey texture.
4. Repeat the previous steps for the other two spheres, defining white_marble for the right sphere and sapphire_agate for the left sphere.

At this point, all three spheres should have textures associated with them. The next step is to create and position light sources. To create a light source, select the Light command from the Misc menu. Enter the following coordinates: X = 5, Y = 5, and Z = –5. Repeat the same process and create a second light source with the following coordinates: X = –5, Y = 5, and Z = –5. In order to view the light sources, you have to zoom out of the scene. To do this, click three times on the out button. The light sources appear as small coordinate systems. Observe that the world definition has changed from (–1, –1, 1, 1) to (–8, –8, 8, 8). Click three times on the in button to return to the original world definition.

The final step in the creation of a scene is the definition of a viewpoint. To create one, select the Viewpoint command from the Misc menu. Figure 7.4 shows what the Viewpoint definition windows looks like. Enter the appropriate numbers in your definition so it looks identical to figure 7.4. After entering the appropriate values, click the radio button titled "include viewpoint." This flags POVCAD to include the current viewpoint definition in the Polyray scene file. To exit the window, click OK.

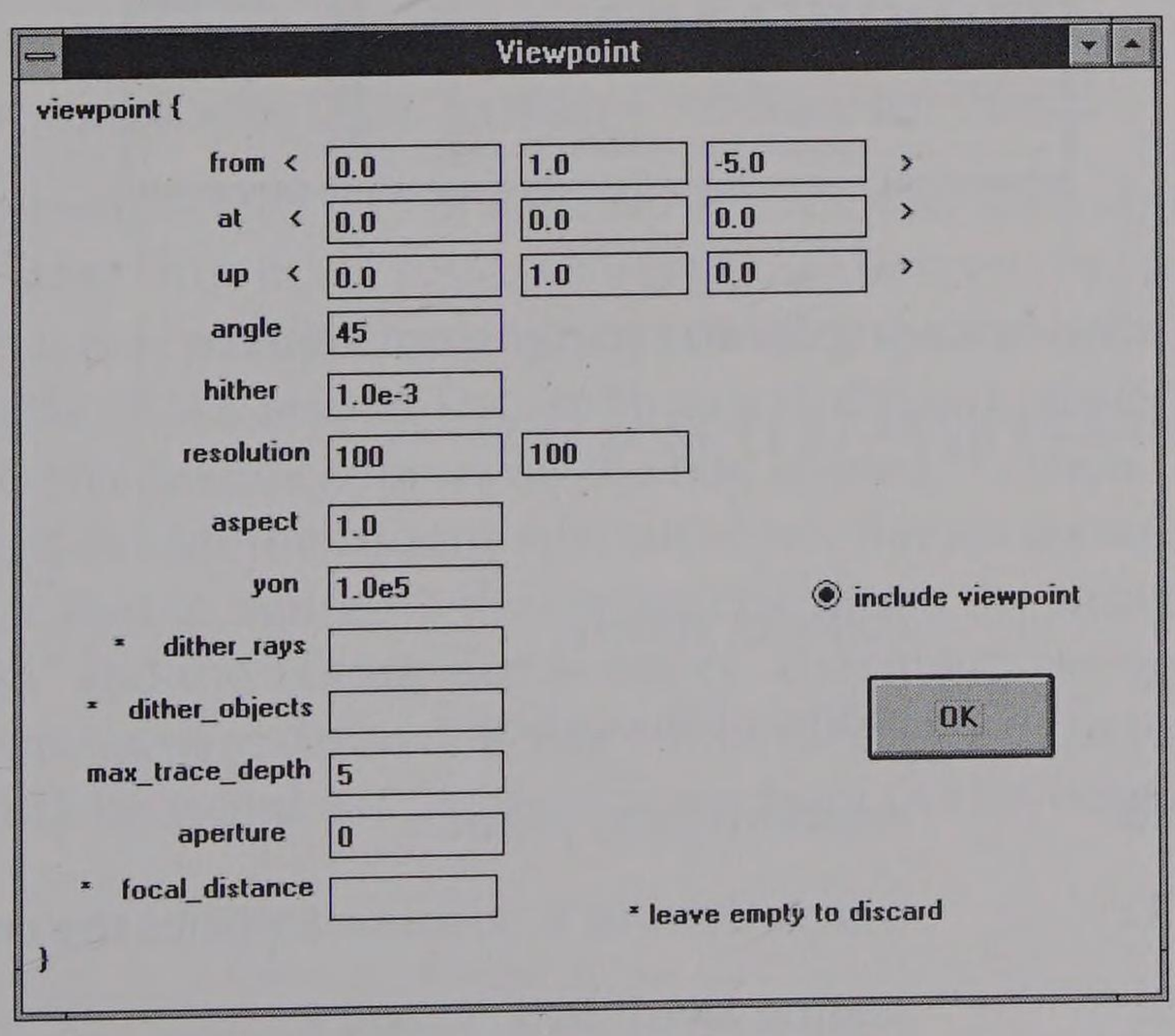


Figure 7.4
Viewpoint screen.

As with any document file or program you are creating, it is important to regularly save the file. To save the current scene, select Save As from the File menu. Type the name "TUTOR1" and press Enter. Notice that the extension was not included, since POVCAD appends the .CAD extension automatically.

The scene has been completed. You can exit POVCAD at any time and return later to continue editing TUTOR1.CAD, but in order to create a Polyray-compatible file, you have to take one additional step. Select the Export command from the File menu. You see two choices, since POVCAD supports both Polyray and the Persistence of Vision (POV) ray tracers. Select Polyray and rename POVCAD.PI as TUTOR1.PI. Press Enter to generate the final Polyray file.

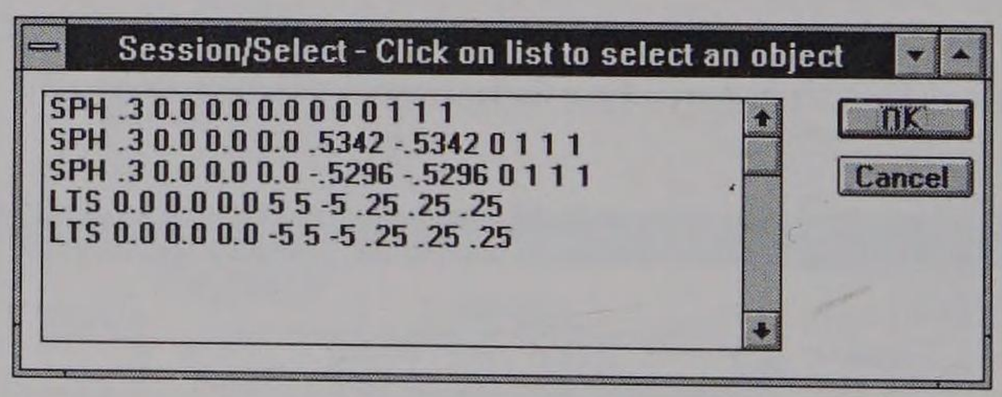
Taking a Closer Look at the TUTOR1.CAD Scene

If you exited POVCAD, return to it and load the TUTOR1.CAD scene by selecting Open from the File menu and clicking on the TUTOR1.CAD file.

POVCAD can display a list of all the objects that compose a given scene. To see this list, click the Session command. Figure 7.5 presents the Session List.

Figure 7.5

POVCAD's Session List for TUTOR1.CAD.



POVCAD uses the following format to represent spheres:

SPH Radius Rx Ry Rz Tx Ty Tz Sx Sy Sz

where

SPH	code for sphere
Radius	radius of the sphere
Rx, Ry, Rz	rotations in x, y, and z
Tx, Ty, Tz	translations in x, y, and z (defines the center point)
Sx, Sy, Sz	scaling factor in x, y, and z

POVCAD uses this format to represent light sources:

LTS 0.0 0.0 0.0 Tx Ty Tz Sx Sy Sz

where

0.0	not used
Tx, Ty, Tz	location of light source
Sx, Sy, Sz	scaling factor in x, y, and z (used only by POVCAD when drawing the light source symbols on the screen). Not used by Polyray.

Now compare the Session List with the content of the files TUTOR1.CAD and TUTOR1.TEX:

Contents of TUTOR1.CAD (geometry definition file)

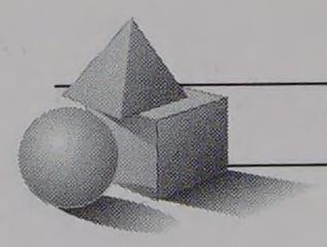
```
"SPH .3 0.0 0.0 0.0 0 0 0 1 1 1"
"SPH .3 0.0 0.0 0.0 .5342 -.5342 0 1 1 1"
"SPH .3 0.0 0.0 0.0 -.5296 -.5296 0 1 1 1"
"LTS 0.0 0.0 0.0 5 5 -5 .25 .25 .25"
"LTS 0.0 0.0 0.0 -5 5 -5 .25 .25 .25"
```

Contents of TUTOR1.TEX (texture definition file)

```
"reflective_grey"
"white_marble"
"sapphire_agate"
""
""
```

Notice first that the content of the TUTOR1.CAD file is identical to the Session List. Then notice that each line in the TUTOR1.TEX file corresponds to the same line in the geometry file (*.CAD). Light Sources did not have any color associated with them so their entries are left blank.

This is a very complicated way of determining whether each object has a texture associated with it. An easier way is to use the scroll bar found at the bottom right corner of the screen. When you click the scroll bar, all of the objects available in the Session List are shown in the box labeled "Object" and the respective texture appears on the box labeled "Texture." If this box is blank, it signals that the current selected object has no texture associated with it. Notice that in figure 7.3 the Object box shows the string "SPH .3 0.0 0.0 0.0 –.5296" and the Texture box is empty. This image was generated before the textures were defined. If you wish to see the rest of the Object box string, click on it and use the right arrow from the keyboard to scroll through it.



The Polyray Scene File

Before you ray trace the image, take a look at the file TUTOR1.PI (see Listing 7.1).

Listing 7.1. First scene created with POVCAD (TUTOR1.PI).

```
//PROGRAM:   TUTOR1.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/1/93 7:21:11 PM

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <0.0, 1.0, -5.0 >
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}

// SPHERE
object {
 sphere < 0.0,0.0,0.0>, 0.3
   reflective_grey
}
// SPHERE
object {
 sphere < 0.5342,-0.5342,0.0>, 0.3
   white_marble
}
// SPHERE
object {
 sphere < -0.5296,-0.5296,0.0>, 0.3
   sapphire_agate
}
// LIGHT_SOURCE
  light < 5 ,  5 , -5  >

// LIGHT_SOURCE
  light <-5 ,  5 , -5  >
```

The file TUTOR1.PI is generated by POVCAD using the files TUTOR1.CAD and TUTOR1.TEX. The objects are generated in the same order they were created. The light sources are at the end of the file. An important point:

The coordinates of the spheres may vary a little from the coordinates that you saw in the Horz and Vert boxes. This is due to the SNAP function being disabled. If you need precise location of objects click on the SnapOff icon to enable it. Coming back to the scene file, another interesting feature is that the viewpoint definition has a lot more information than you used in the previous chapters. This is the viewpoint generated (with my modifications):

```
viewpoint {
from      <0.0, 1.0, -5.0 >
at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 //hither     1.0e-3
 resolution 100,100
 aspect     1.0
 //yon        1.0e5
 //max_trace_depth 5
 //aperture        0
}
```

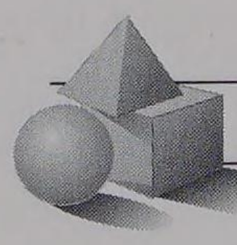
Normally, I delete `hither, yon, max_trace,` and `aperture` (it's a personal choice; you don't have to). You can also comment them out using the `//` symbol as shown. This brings up an important point: Is the scene perfect once it is generated by POVCAD? The answer is, "Maybe." If you ray trace the image as it comes out you will not have any syntax errors, but the definition of the viewpoint could be improved. The objects are not centered and they look rather small. The screen is not used completely. You'll have to modify the file by hand using a text editor. After some experimentation, I made two additional changes to center and maximize the size of the objects:

```
from      <0.0, 1.0, -2.0 >
at        <0.0, -0.2, 0.0 >
```

If you include these two changes, the objects will be in the center of the screen and their size will be as large as possible.

Before you ray trace this scene, remember to copy the TUTOR1.PI file to the directory where POLYRAY resides or else you will get errors.

Figure 7.6 shows the final ray-traced TUTOR1.PI image.



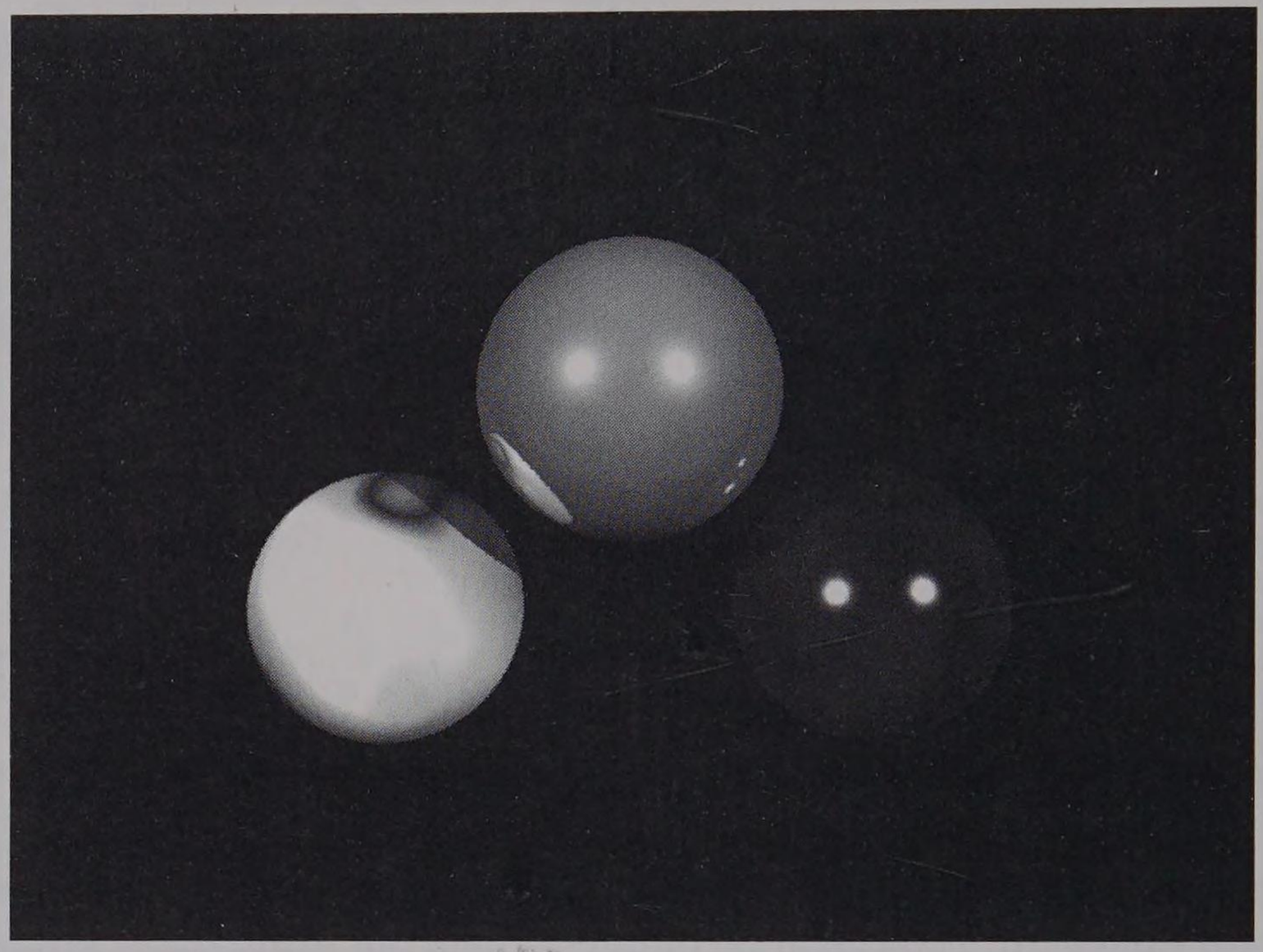


Figure 7.6

TUTOR1.PI ray-traced image.

Using POVCAD Primitives (Tutorial #2)

In this session, you create a scene using a few of POVCAD's primitives. First, you create a plane to use as a floor. Then you create a sphere, an ellipsoid, a box, and a cone, each with a different texture or color. Finally, you define the light sources and viewpoint.

Floor:

1. From the Object menu, select Plane_Y.
2. Select Xform and type the following values, clicking OK when done:

   Scale x = 100, y = 100, and z = 100

3. Click Texture and select Stone1

At this point, the plane is not visible, since the world limits are (–1, –1, 1, 1) and the plane has limits in (–100, –100, 100, 100).

Sphere:

1. From the Object menu, select Sphere.

2. Enter the following values, pressing Enter after each input:

Radius	0.5
x	0.0
y	0.5
z	0.0

3. Double-click the out button to zoom out.
4. Select Texture and pick reflective_cyan. You don't have to select the sphere since it's the last object you created.

Ellipsoid:

1. From the Object menu, select Ellipsoid.
2. Enter the following values (press Enter after each input):

x Radius	1.0
y Radius	0.5
z Radius	0.5

3. Click Xform and type the following values, clicking OK when done:

Rotate	y 45.0
Trans	x 1.5
Trans	y 0.5

4. Select Texture, Stone13.

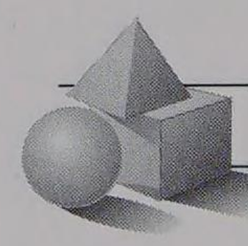
Box:

1. From the Object menu, select Box.
2. Enter the following dimensions (press Enter after each input):

x Length	0.5
y Length	0.5
z Length	2.0

3. Select Xform.

Trans	x –1.5
Trans	y 0.5



4. Texture menu, reflective_gold.

Cone:

1. From the Object menu, select Cone.
2. Enter the values:

Radius	0.5
Height	1.0

3. Xform menu:

Trans	y	1.0
Trans	z	–2.0

4. Select the Colors command, SkyBlue.

Torus:

1. Select Torus from the Object menu.
2. Enter the values:

Minor Radius	0.3
Major Radius	4.0

3. Xform menu:

Trans	y	0.3

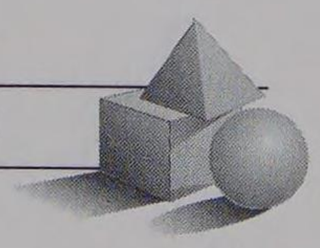
4. Texture menu, reflective_white.

At this point you should save the file, in case something goes wrong. To save the current session, from the File menu select the Save As command. Enter the name TUTOR2 and press the Enter key. The only objects missing on the scene are the light sources and the viewpoint definition.

Light Source:

1. From the Misc menu, select Light.
2. Enter the location:

x position	–10.0
y position	5.0
z position	–10.0



Light Source:

1. Create another light source and enter the location:

x position	10.0
y position	5.0
z position	–10.0

Save the session again, this time use the Save command (in the File menu) since the session already has a name. Figure 7.7 shows the screen displayed in POVCAD.

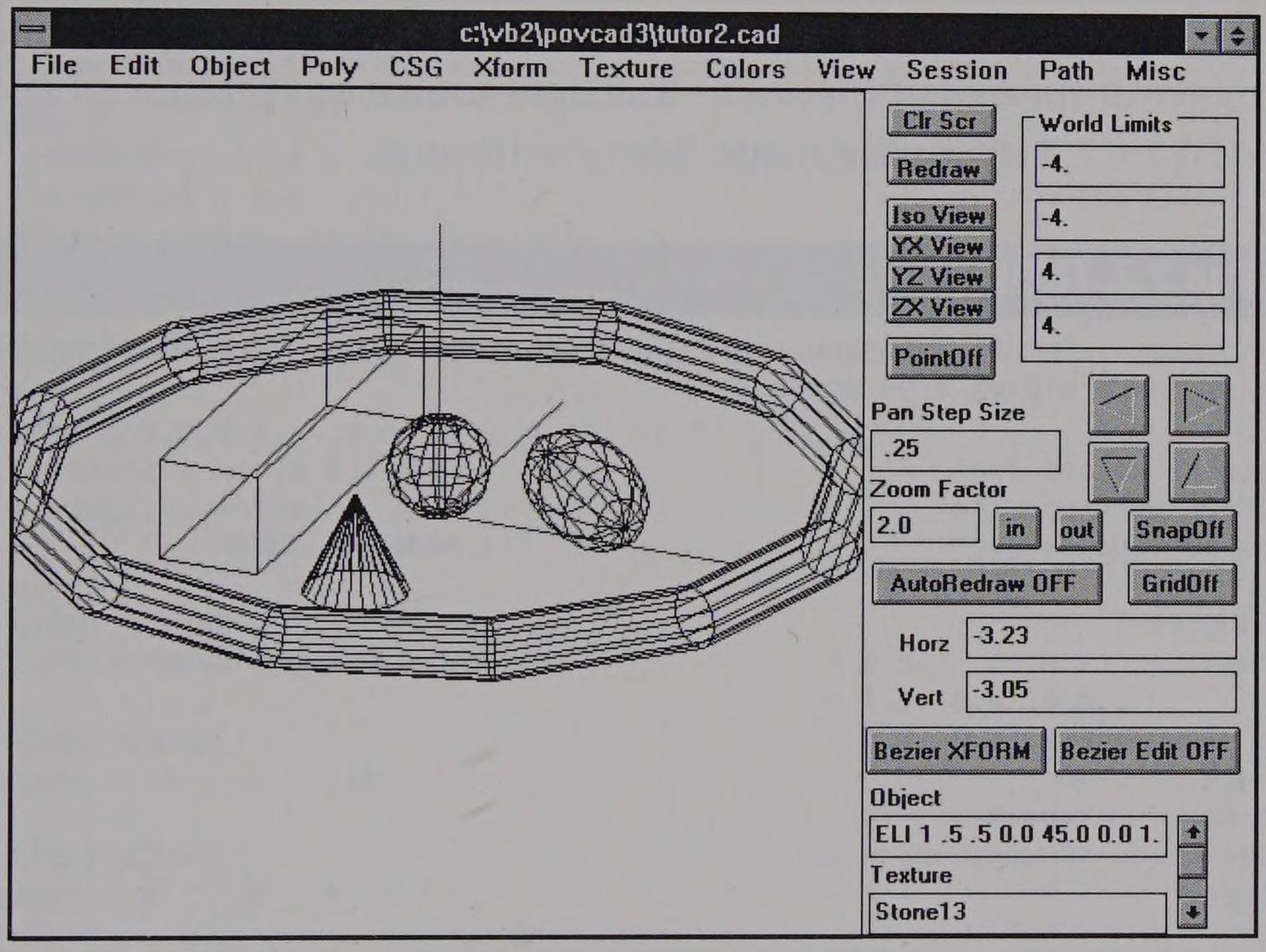


Figure 7.7

The TUTOR2 session in POVCAD.

Now create the viewpoint:

1. From the Misc menu, select viewpoint.
2. Enter the following parameters:

from	<0, 5, –10>
resolution	100,100
aperture	0

3. Click the radio button labeled "include viewpoint."
4. To leave, click OK.

You don't have to save the scene again, since the viewpoint information is not saved by POVCAD. The final step is to create a Polyray file using the Export command. From the File menu, select Export and then Polyray. Replace the name POVCAD.PI with TUTOR2.PI and press Enter or click OK. You can exit POVCAD at this point.

If you try ray tracing TUTOR2.PI, Polyray reports that there is an error in line 25. A closer look at TUTOR2.PI (see Listing 7.2) shows that Polyray does not understand the texture name Stone1. This occurs because the file `"stones.inc"` was not included at the top of the scene file. After including it, I ray traced the image and found that the scene could use more light and the viewpoint was too near for my taste (you may have a different opinion). The next step was to edit the viewpoint definition and add an additional light source at the end of the scene. The light source was positioned at <0, 30, 0>. Figure 7.8 shows the results after ray tracing.

Listing 7.2. TUTOR2.PI.

```
//PROGRAM:   TUTOR2.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/2/93 4:20:03 PM

include "colors.inc"
include "texture.inc"
include "stones.inc"                          // Added by hand!

viewpoint {
 from      <2.0, 5, -12.0 >                   // Modified by hand!
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}
// PLANE (Polygon)
object {
// normal in Y direction
  polygon 4, <-1,0,1>, <-1,0,-1>, <1,0,-1>, <1,0,1>
    scale < 100.0,100.0,100.0>
   Stone1
}

// SPHERE
object {
 sphere < 0.0,0.5,0.0>, 0.5
```

```
   reflective_cyan
}
// ELLIPSOID
object {
 sphere <0.0, 0.0, 0.0>, 1
   scale <1.0, 0.5, 0.5>
    rotate < 0.0,45.0,0.0>
    translate < 1.5,0.5,0.0>
   Stone13
}
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 0.5,0.5,2.0>
    translate < -1.5,0.5,0.0>
   reflective_gold
}

// CONE
object {
    cone <0.0, -1.0,0.0>, 0.5, <0.0, 0.0, 0.0>, 0.0
    translate < 0.0,1.0,-2.0>
 texture { surface {color SkyBlue}}
}

// TORUS
object {
    torus 4.0,0.3, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    translate < 0.0,0.3,0.0>
   reflective_white
}

// LIGHT_SOURCE
  light <-10 ,  5 , -10  >

// LIGHT_SOURCE
  light < 10 ,  5 , -10  >

// LIGHT_SOURCE                              // Added by hand!
  light < 0 ,  30 , 0  >
```

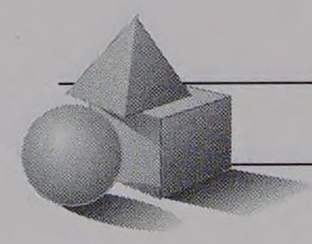
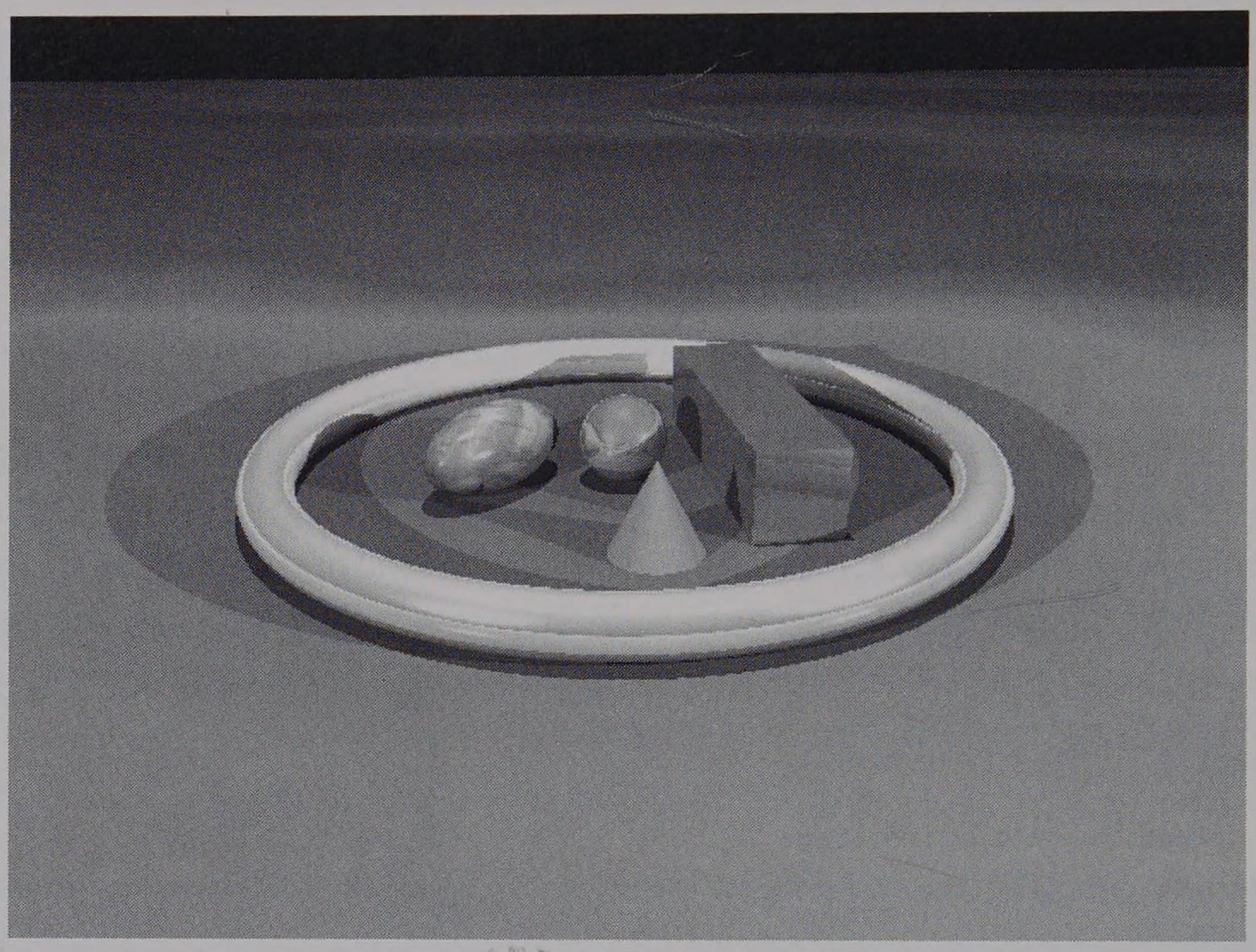


Figure 7.8

Ray trace of TUTOR2.PI.



Creating Multiple Copies of Objects (Tutorial #3)

Looking back at figure 3.7, you can see the usefulness of having a function that creates multiple copies of an object. In this session, you have the opportunity to create an image similar to figure 3.7.

Center Torus:

1. Select Torus from the Object menu, and enter the values:

   minor radius 0.1

   major radius 1.0

2. To zoom out, double-click the out button.
3. Select Xform:

   Rotate X 90

4. Select the YX View.
5. Select Texture, reflective_blue.

6. From the Object menu select Torus, enter the values:

   minor radius 0.1

   major radius 1.0

7. Click the SnapOff button (it should flip to SnapOn).

8. Click the GridOff button (it should flip to GridOn).

   At this point, the grid should appear on-screen. The Snap function forces the mouse to select only points which are located at the intersection of the grid lines. This method is used to select precise locations.

9. Locate the mouse at Horz = 1.6 and Vert = 0.0. Press the right mouse button.

10. Select Move (the second torus should move to the new position).

    You have created the center torus and one of the tori which will be copied and rotated through 330 degrees. In order to apply the rotation follow the next steps.

11. Move the mouse to the origin Horz = 0.0 and Vert = 0.0 (this will be your pivot point for the rotation). Press the right mouse button.

12. Select Copy/Rotate and enter the following values (Figure 7.9 shows the image that should appear on your screen.):

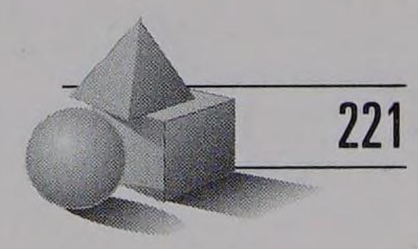
    rotation angle 30

    # of copies 11

13. If you wish to see other views of the image do so now by clicking on Iso View, YZ View, or ZX View. When done, return to YX View.

    The next step is to attach textures to each torus. You do this by simply selecting each torus and then defining the texture. To select a torus, click the mouse near the center of the torus. The torus selected will change colors.

14. Select one torus and attach a texture. Since the first torus already has a texture, you might want to start on torus #2 (the first copy). These are the textures I used for each one (all textures are of the type reflective_?):



1	blue	6	green	11	white
2	brown	7	grey	12	yellow
3	coral	8	orange	13	blue
4	cyan	9	red		
5	gold	10	tan		

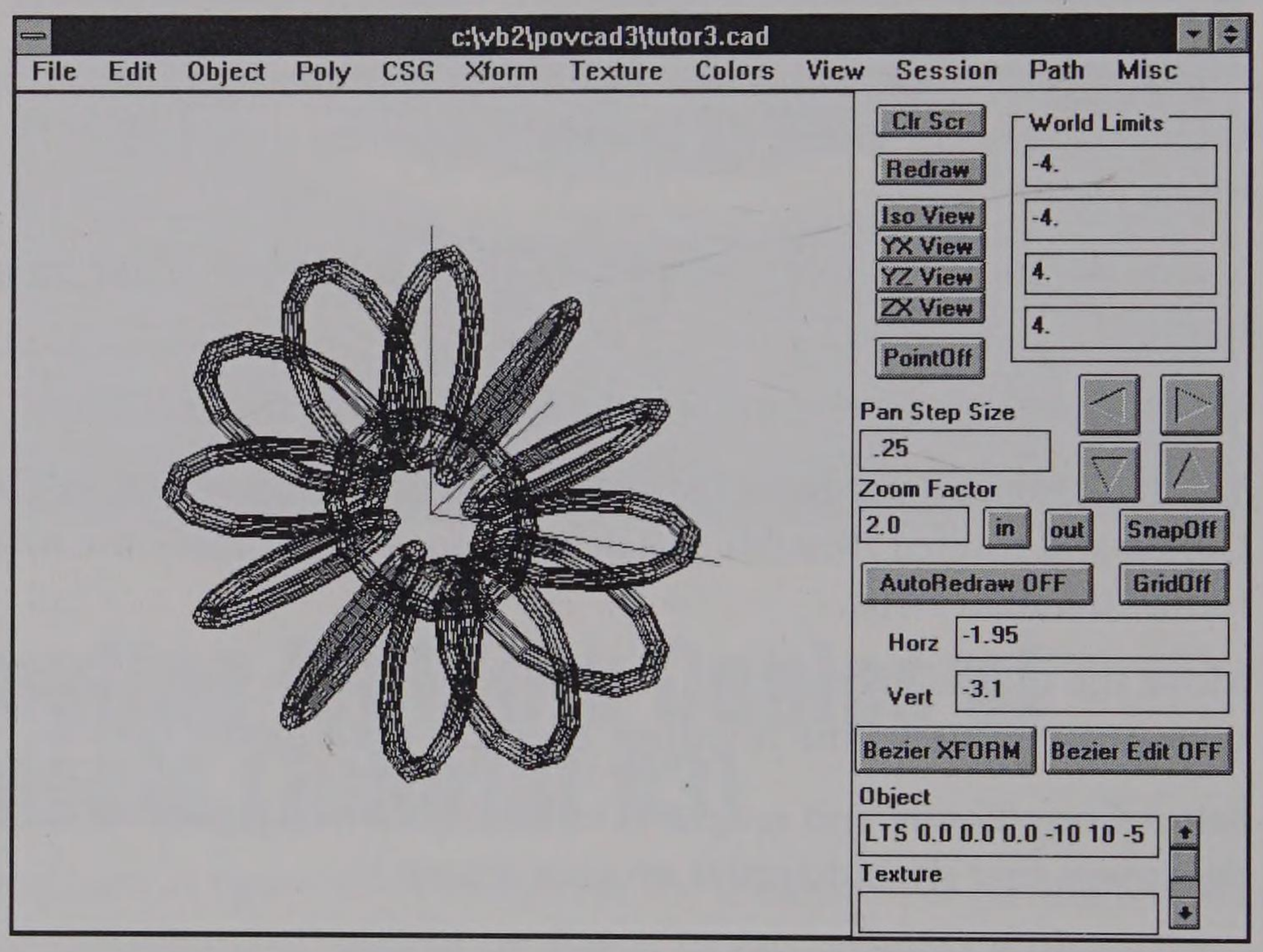


Figure 7.9

POVCAD's display after creating multiple copies of the tori (Iso View, Grid Off).

15. To check that each torus has a texture, use the scroll bar located at the lower right corner of the screen. To go back to the first object, press the scroll bar button with the arrow pointing up. If the texture box is empty, stop and select a texture for that object. Another way of checking the textures is to click on each one and check the texture box to see if it's empty. If it is, select a texture for it.

16. Save the image using the Save As command. Use the name TUTOR3 and press the Enter key.

Create two light sources with the following locations:

17. Light source #1 located at <5, 0, -10>.
18. Light source #2 located at <-10, 10, -5>.
19. Use the Save command to save the light source information.
20. Create the viewpoint (remember to click "include viewpoint"):

from	<1, 3, –5>
resolution	100,100
aperture	0

21. Generate the Polyray file using Export. Save the file as TUTOR3.PI.

Listing 7.3 presents the file generated by POVCAD for the TUTOR3 session. The viewpoint from vector was modified slightly to have a better view of the scene. Figure 7.10 shows the final ray-traced image.

Listing 7.3. TUTOR3.PI Polyray Scene File.

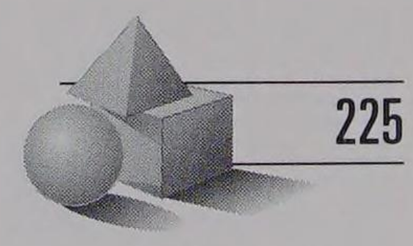
```
//PROGRAM:   TUTOR3.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/2/93 9:45:15 PM

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <1.0, 3.0, -7.0 >          // Modified by hand!
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 90.0,0.0,0.0>
   reflective_blue
}
```

continues

Listing 7.3. Continued.

```
// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    translate < 1.6,0.0,0.0>
   reflective_brown
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,30.0>
    translate < 1.385641,0.8,0.0>
   reflective_coral
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,60.0>
    translate < 0.8,1.385641,0.0>
   reflective_cyan
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,90.0>
    translate < 0.0,1.6,0.0>
   reflective_gold
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,120.0>
    translate < -0.8,1.385641,0.0>
   reflective_green
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,150.0>
    translate < -1.385641,0.8,0.0>
   reflective_grey
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,180.0>
    translate < -1.6,0.0,0.0>
   reflective_orange
}
```

```
// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,210.0>
    translate < -1.385641,-0.8,0.0>
   reflective_red
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,240.0>
    translate < -0.8,-1.385641,0.0>
   reflective_tan
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,270.0>
    translate < 0.0,-1.6,0.0>
   reflective_white
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,300.0>
    translate < 0.8,-1.385641,0.0>
   reflective_yellow
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 0.0,0.0,330.0>
    translate < 1.385641,-0.8,0.0>
   reflective_blue
}

// LIGHT_SOURCE
  light < 5 ,  0 , -10  >

// LIGHT_SOURCE
  light <-10 ,  10 , -5  >
```



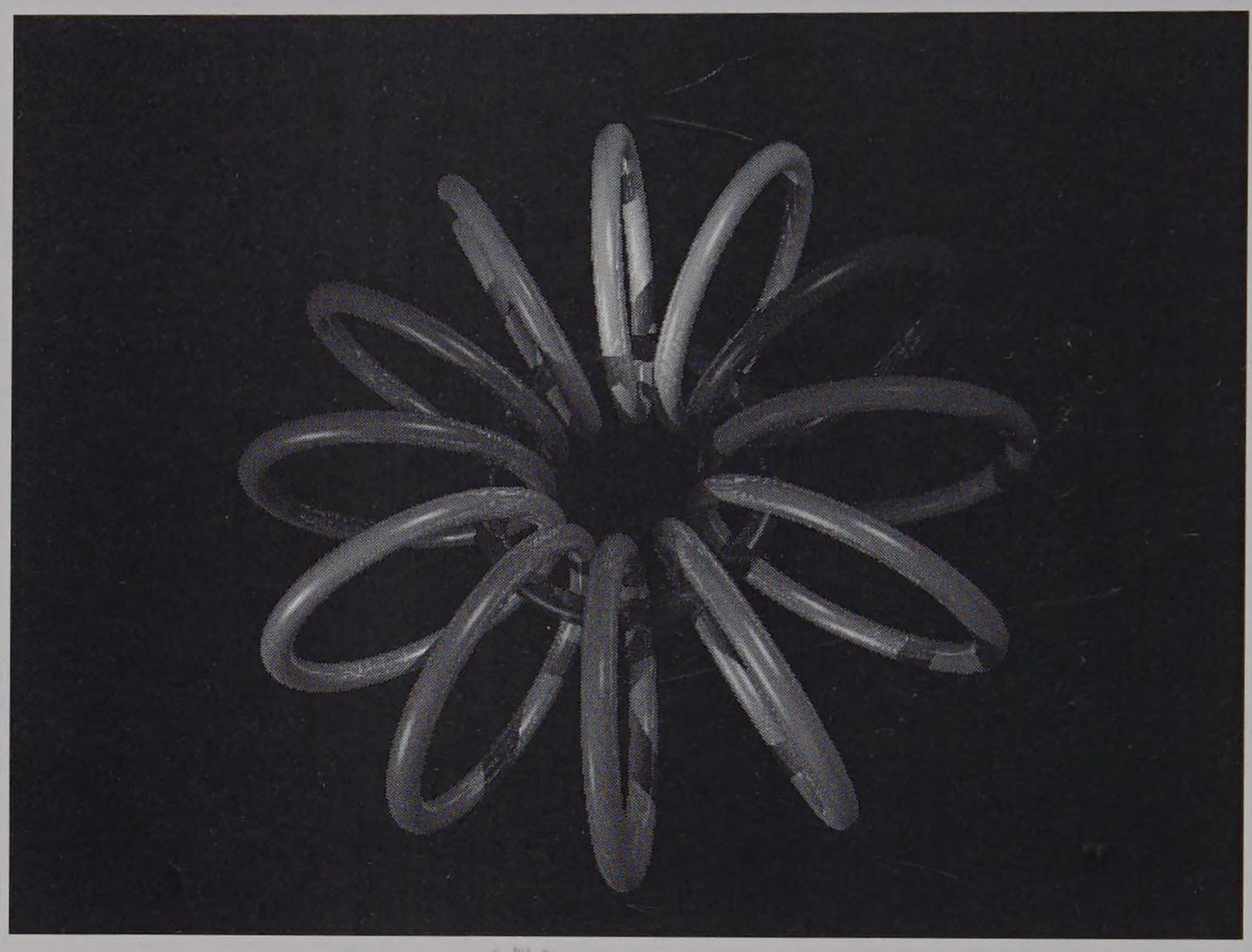


Figure 7.10
Ray-traced image of TUTOR3.PI.

You can simplify the file TUTOR3.PI generated by POVCAD if you consider that each torus can be created by a simple rotation statement. You can use the second torus you created and convert it into a primitive. Then all the remaining objects can be created by rotating the primitive. Listing 7.4 presents TUTOR3M.PI, a modified version of TUTOR3.PI.

Remember the steps taken to create the rotating torus objects. First, create a torus and displace it in the x axis by 1.6 units. Then, select the pivot point for the rotation at the origin. Select the Copy/Rotate command by pressing the right mouse button. The rotation angle is 30 degrees and the number of copies to make is 11. Therefore, every 30 degrees a torus is created (a total of 11 copies, not counting the original).

From this, we can create a primitive called OuterTorus located at <1.6, 0, 0>.

```
define OuterTorus
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    translate < 1.6,0.0,0.0>
}
```

The first (original) and the second copy would be

```
// TORUS
object {
```

```
   OuterTorus
   reflective_brown
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,30.0>
   reflective_coral
}
```

In other words, the object definition could be generalized as

```
// TORUS
object {
   OuterTorus
   rotate < 0.0, 0.0, Rx >
   ...texture...
}
```

where *Rx* varies from 0 to 330 degrees.

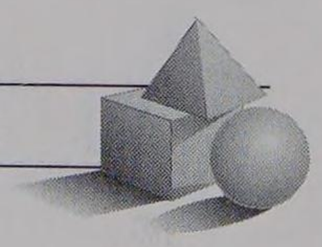
Listing 7.4. Improving TUTOR3.PI by defining a primitive.

```
// TUTOR3M.PI
// HAND SIMPLIFIED VERSION

include "colors.inc"
include "texture.inc"


define OuterTorus
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    translate < 1.6,0.0,0.0>
}

viewpoint {
 from      <1.0, 3.0, -7.0 >          // Modified by hand!
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}

// TORUS
object {
    torus 1.0,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 90.0,0.0,0.0>
   reflective_blue
}
```

continues

Listing 7.4. Continued.

```
// TORUS
object {
   OuterTorus
   reflective_brown
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,30.0>
   reflective_coral
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,60.0>
   reflective_cyan
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,90.0>
   reflective_gold
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,120.0>
   reflective_green
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,150.0>
   reflective_grey
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,180.0>
   reflective_orange
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,210.0>
   reflective_red
}

// TORUS
```



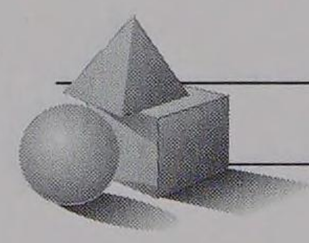
```
object {
   OuterTorus
   rotate < 0.0,0.0,240.0>
   reflective_tan
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,270.0>
   reflective_white
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,300.0>
   reflective_yellow
}

// TORUS
object {
   OuterTorus
   rotate < 0.0,0.0,330.0>
   reflective_blue
}

// LIGHT_SOURCE
  light < 5 ,  0 , -10  >

// LIGHT_SOURCE
  light <-10 ,  10 , -5  >
```

Both TUTOR3.PI and TUTOR3M.PI define the same geometry. POVCAD is helpful in the sense that it lets you create prototypes quickly. You will have many cases like this, where you will decide that the scene can be made simpler and more readable by modifying it yourself.

Using CSG Operations (Tutorial #4)

POVCAD supports the use of Boolean operations to create primitives. The following session creates an object we'll call Elbow. Elbow is created by intersecting a torus and a box. Two cylinders are attached at the ends of the torus section. The three objects together have a shape similar to a piece of metal tubing bent 90 degrees. The bend is round and smooth. Creating this object requires two Boolean operations: intersection (*) and union (+), which can be written as

```
(torus * box) + cylinder#1 + cylinder#2
```



After you create the Elbow primitive, you'll develop a second scene by hand that will use the Elbow extensively.

POVCAD follows a special approach in order to define Boolean operations. As an example, we'll take the expression `(torus * box) + cylinder#1 + cylinder#2` and explain how it will be developed using the program. First, the sub-expression `(torus * box)` is composed of two objects but the result is just one: a section of the torus. You can then simplify the initial expression to

```
torus_section  + cylinder#1 + cylinder#2
```

where `torus_section` is (`torus * box`). POVCAD uses the format

```
UNION
torus_section
cylinder#1
cylinder#2
ENDF
```

to represent the operation (this is shown in the Session command when the scene is created). This can be described as "create a union of the objects torus_section, cylinder#1, and cylinder#2." The expression `ENDF` indicates that the `UNION` operation is completed, in other words `ENDF` flags POVCAD that this CSG operation is complete—ENDFinal.

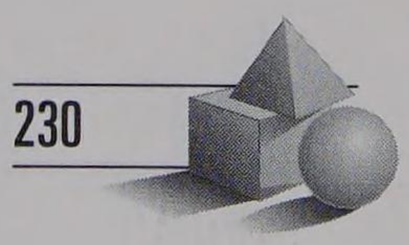
The next step is to work on the expression torus_section, which we said was the same as `(torus * box)`. POVCAD represents this operation as

```
INTRS
torus
box
ENDF
```

This can be described as "create an intersection between a torus and a box." The expression `INTRS` represents the intersection operation. If you substitute one into the other, you have

```
UNION
INTRS
torus
box
ENDF
cylinder#1
cylinder#2
ENDF
```

POVCAD understands that the first `ENDF` closes the `INTRS` operation and not the `UNION`, so it replaces it with `END`. In other words, `END` is used to complete partial CSG operations while `ENDF` denotes that the complete CSG operation is now finished.




```
UNION
INTRS
torus
```

```
box
END
cylinder#1
cylinder#2
ENDF
```

This is roughly the format used by POVCAD to generate a CSG object. Take a look at the TUTOR4.CAD file that you will be creating shortly:

```
"UNION"
"INTRS"
"TOR .2 .1 90.0 0.0 0.0 0.5 0.5 0.0 1 1 1"
"BOX 0.0 0.0 0.0 0.7 0.7 0.0 .2 .2 .2"
"END"
"CYL .1 .5 0.0 0.0 90.0 0.25 0.7 0.0 1 1 1"
"CYL .1 .5 0.0 0.0 0.0 0.7 0.25 0.0 1 1 1"
"ENDF"
"LTS 0.0 0.0 0.0 5 5 -5 .25 .25 .25"
"LTS 0.0 0.0 0.0 1 5 -5 .25 .25 .25"
```

Observe that it follows the same logic that was just explained. The main point to remember is to create the objects and select CSG commands in this same order. The command to create the `END` and `ENDF` words is Close and can be found in the CSG menu. The CSG menu has the following commands:

CSG Menu:

- Union
- Difference
- Intersection
- Composite (not used in Polyray)
- Close

Based on the TUTOR4.CAD output, here are the steps to create such a scene in POVCAD:

Session List Display	Operation Performed By User
UNION	CSG menu, click Union
INTRS	CSG menu, click Intersection
TOR ...	Object menu, click Torus
BOX ...	Object menu, click Box
END	CSG menu, click Close
CYL ...	Object menu, click Cylinder

continues

Session List Display	Operation Performed By User
CYL ...	Object menu, click Cylinder
ENDF	CSG menu, click Close
LTS ...	Misc menu, click Light
LTS ...	Misc menu, click Light

The sequence is intuitive, and after this tutorial you will be able to create objects with any amount of complexity. The following session will take you through the complete steps. (The parameters following the keywords were omitted for simplicity.)

To start constructing the scene, select the YX View and follow these steps:

1. CSG menu, select Union
2. CSG menu, select Intersection
3. Objects menu, Torus:

   Minor Radius 0.1

   Major Radius 0.2
4. Xform:

   Rotate x 90

   Trans x 0.5

   Trans y 0.5
5. Object menu, Box:

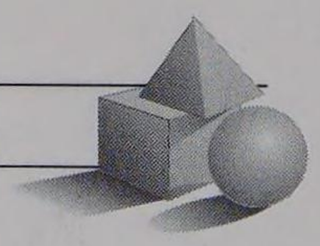
   x Length 0.2

   y Length 0.2

   z Length 0.2
6. Xform:

   Trans x 0.7

   Trans y 0.7



7. CSG menu, select Close (this step closes the intersection operation but the union remains open).

Now, let's create the cylinders:

1. Object menu, Cylinder

Radius	0.1
Length	0.5

2. Xform:

Rotate z	90
Trans x	0.25
Trans y	0.7

3. Object menu, Cylinder

Radius	0.1
Length	0.5

4. Xform:

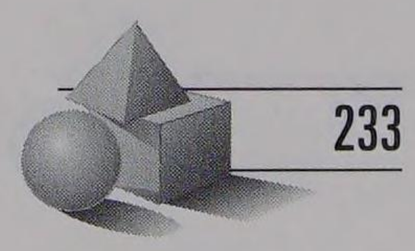
Trans x	0.7
Trans y	0.25

5. CSG menu, select Close (This step closes the union operation.)

   To define a texture for the object as a group, select the Sessions command from the main menu and pick ENDF. Click Texture and choose reflective_gold.

6. Save the session as TUTOR4 with the Save As command.

Figure 7.11 displays the screen generated by POVCAD.



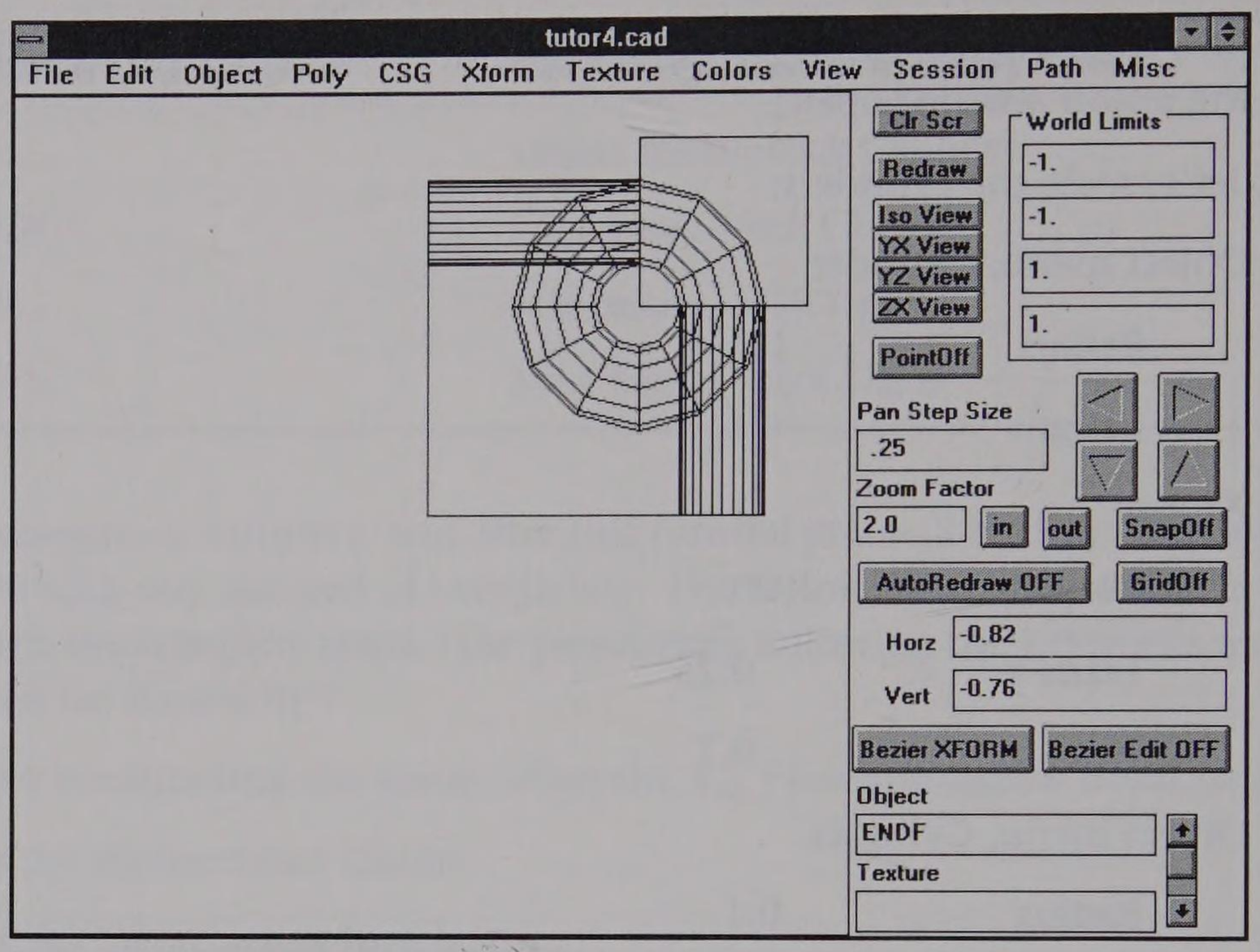


Figure 7.11

Display of the Elbow primitive in POVCAD.

As always, create the light sources and the viewpoint:

1. Light source #1 located at <5, 5, –5>.
2. Light source #2 located at <1, 5, –5>.
3. Define viewpoint:

from	<0.5, 0.5, –2>
at	<0.5, 0.5, 0.0>
resolution	100, 100
aperture	0

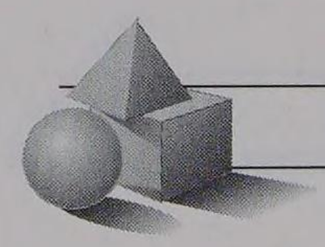
Also click on "include viewpoint."

To complete the session, use the Save command, then Export to a Polyray file: TUTOR4.PI. Figure 7.12 is the ray-traced image of TUTOR4.PI. Listing 7.5 shows TUTOR4.PI.

Listing 7.5. Using CSG operations (TUTOR4.PI).

```
//PROGRAM:   TUTOR4.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/3/93 9:52:34 AM

include "colors.inc"
include "texture.inc"

viewpoint {
 from       <0.5, 0.5, -2.0 >
 at         <0.5, 0.5, 0.0 >
 up         <0.0, 1.0, 0.0 >
 angle       45
 hither      1.0e-3
 resolution 100,100
 aspect      1.0
 yon         1.0e5
 max_trace_depth 5
 aperture        0
}

// CSG START
object {
// TORUS
object {
    torus 0.2,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 90.0,0.0,0.0>
    translate < 0.5,0.5,0.0>

}

*
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 0.2,0.2,0.2>
    translate < 0.7,0.7,0.0>

}

+
// CYLINDER
object {
    cylinder <0.0, -0.25,0.0>, <0.0, 0.25, 0.0>, 0.1
    rotate < 0.0,0.0,90.0>
    translate < 0.25,0.7,0.0>

}

+
// CYLINDER
object {
    cylinder <0.0, -0.25,0.0>, <0.0, 0.25, 0.0>, 0.1
    translate < 0.7,0.25,0.0>

}
```

continues

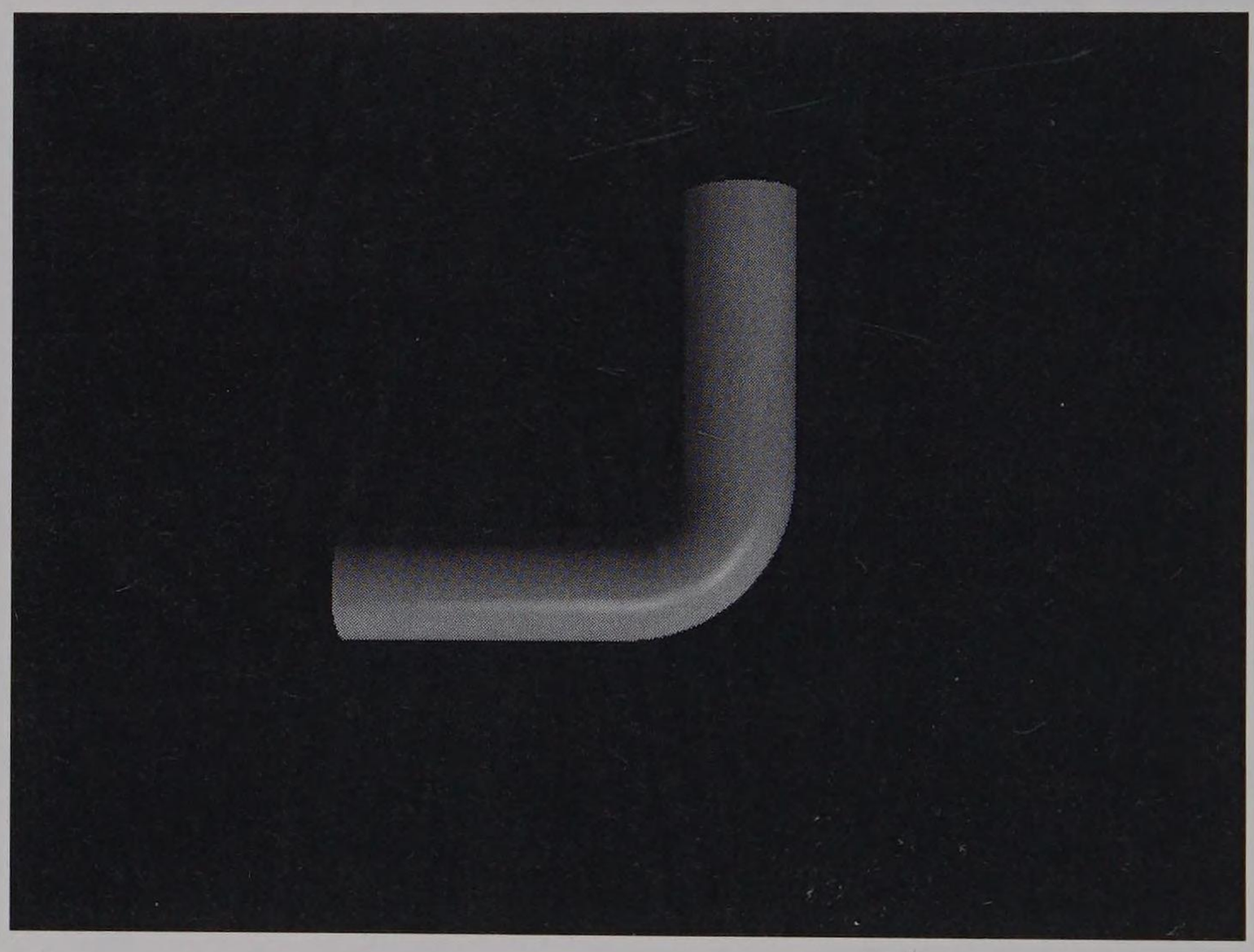


Listing 7.5. Continued.

```
   reflective_gold
}
// CSG END

// LIGHT_SOURCE
  light < 5 ,  5 , -5  >

// LIGHT_SOURCE
  light < 1 ,  5 , -5  >
```

Figure 7.12

Ray-traced Elbow primitive from TUTOR4.PI.



The scene file TUTOR4.PI only creates the Elbow primitive. Now that it's available, we can use it to develop more complex objects. Listing 7.6 presents a scene (TUTOR4M.PI) which expands on the Elbow object. The scene consists of a plane made of wood, with a tube on top of it that has been bent four times—roughly resembling the letter "P." This is done by making multiple copies of the Elbow.

Listing 7.6. Creating a letter "P" (TUTOR4M.PI).

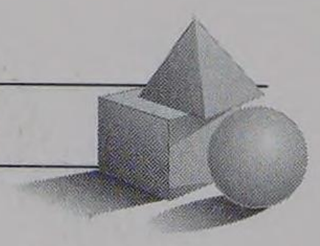
```
//TUTOR4M.PI
//HAND MODIFIED

include "colors.inc"
include "texture.inc"
```

```
background grey

define Plane_XZ  object {
                  polygon 4, <-5,0,5>, <-5,0,-5>, <5,0,-5>, <5,0,5>
                 }
//background grey


viewpoint {
 from      <1.0, 3, -9.0 >
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}

// LIGHT_SOURCE
  light < 5 ,  5 , -5  >

// LIGHT_SOURCE
  light < 1 ,  5 , -5  >


define Elbow                          // Create A Primitive

// CSG START
object {
// TORUS
object {
    torus 0.2,0.1, < 0.0, 0.0, 0.0>, <0.0, 1.0, 0.0>
    rotate < 90.0,0.0,0.0>
    translate < 0.5,0.5,0.0>
}

*
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 0.2,0.2,0.2>
    translate < 0.7,0.7,0.0>
}

+
// CYLINDER
object {
    cylinder <0.0, -0.25,0.0>, <0.0, 0.25, 0.0>, 0.1
    rotate < 0.0,0.0,90.0>
    translate < 0.25,0.7,0.0>
}

+
// CYLINDER
```

continues

Listing 7.6. Continued.

```
object {
    cylinder <0.0, -0.25,0.0>, <0.0, 0.25, 0.0>, 0.1
    translate < 0.7,0.25,0.0>
}

   reflective_gold
}
// CSG END


define DoubleElbow          // Create Another Primitive
object {
 object {
   Elbow
 }
 object {
   Elbow
   rotate <180, 0, 0>
 }
 reflective_gold
}

// Construct A New Object

object {
   Elbow
}

object {
  Elbow
  rotate<0, 180, 0>
  translate<-1, 0, 0>
}

object {
    cylinder <0.0, -0.5, 0.0>, <0.0, 0.5, 0.0>, 0.1
    rotate <0, 0, 90>
    translate < -0.5, 0.7, 0.0>
    reflective_gold
}

object {
    cylinder <0.0, -1.0, 0.0>, <0.0, 1.0, 0.0>, 0.1
    translate < -1.7, -0.7, 0.0>
    reflective_gold
}

object {
   DoubleElbow
   rotate<0, 0, -90>
}
```



```
object {
   Plane_XZ
   translate<0, -3, 4>
   wooden
}
```

In TUTOR4M.PI, two primitives are created: Elbow, which is the object created on TUTOR4.PI, and DoubleElbow, which is the Elbow object in union with another copy of itself, rotated 180 degrees about the x axis. It resembles an inverted letter "C":

```
define DoubleElbow
object {
 object {
   Elbow
 }
+
 object {
   Elbow
   rotate <180, 0, 0>
 }
 reflective_gold
}
```

Finally, figure 7.13 presents the ray-traced scene.

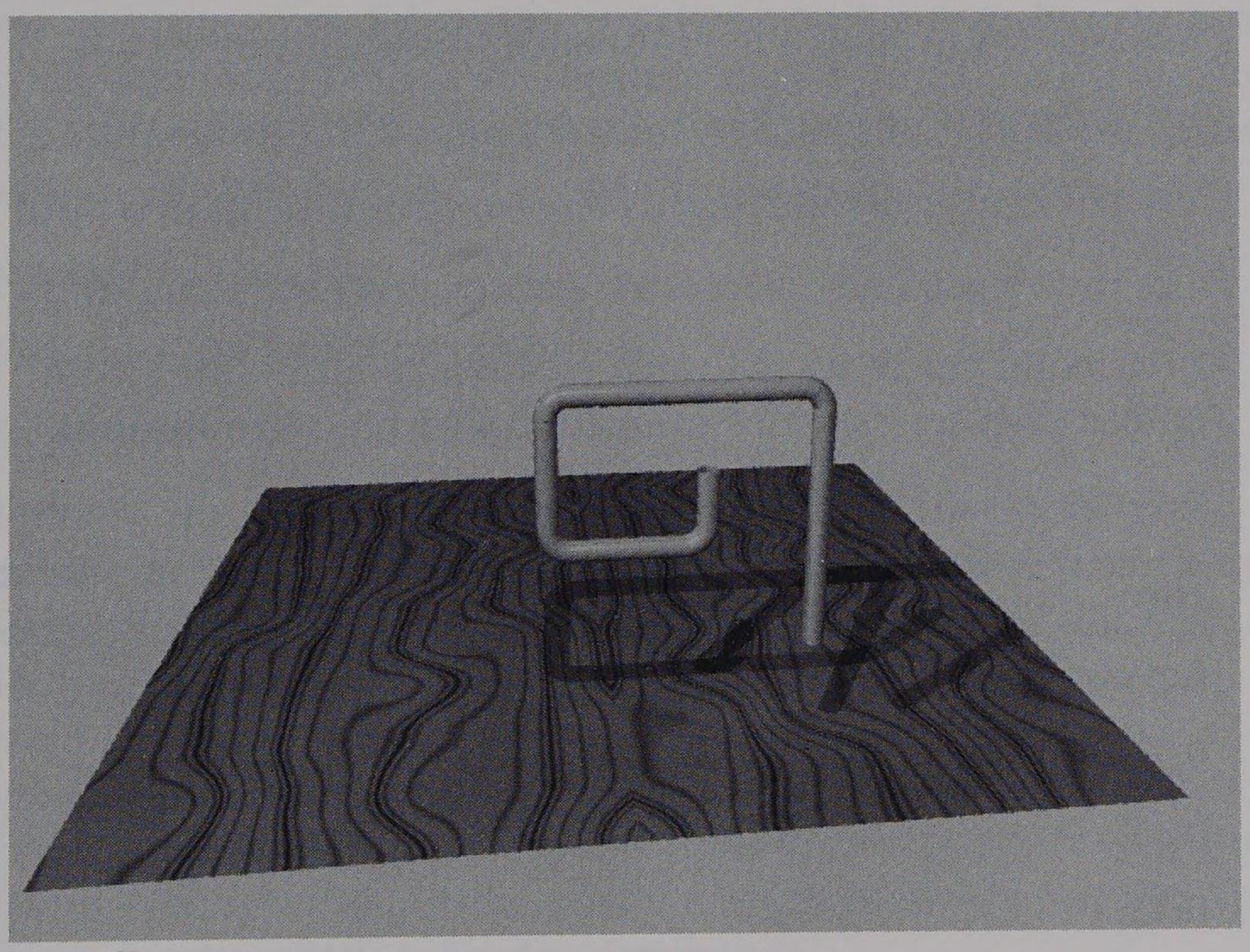


Figure 7.13
Bent tubing forming a letter "P" (TUTOR4M.PI).

Sweep and Extrude Objects (Tutorial #5)

POVCAD can create Sweep and Extrude objects. In Polyray, a Lathe object is the same as a Sweep object in POVCAD. Extrude objects are the same in both. For convenience, I'll use Sweep when referring to Lathe objects, and I'll call Extrude objects by their name.

In this session, you draw contours in POVCAD and rotate them 360 degrees to create a Sweep object. In addition, you Extrude a curve in the z axis. Both objects are created with two different methods: using POVCAD to generate flat triangles, and through the use of Polyray's Lathe and Extrude objects. As you see at the end of the session, there is no comparison between POVCAD's and Polyray's method. POVCAD generates a very rough object since the triangles are flat, but Polyray smooths the surface. To create the Lathe object, you use the program LATHE.BAS presented in Chapter 3. The Sweep object in Polyray is created by hand.

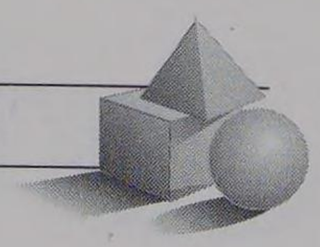
Let's start by creating the Sweep object:

1. Start POVCAD and click the YX View.
2. Click SnapOff (it should flip to SnapOn—Snap feature is ON).
3. Click GridOff (it should flip to GridOn—Grid feature is ON).
4. Click PointOff (it should flip to PointOn—Point creation is ON).
5. From the View menu, select Grid Spacing. Enter the value 40.

   At this point you're ready to start creating points on the screen. The grid lines are increased from 20 to 40, to increase the detail.

6. Click the following locations on the screen:

Horz	Vert
0.2	0.75
0.15	0.65
0.15	0.55
0.35	0.35
0.4	0.15



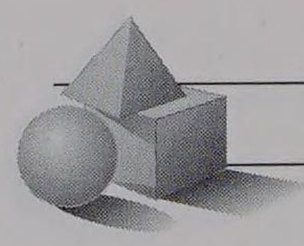
Horz	Vert
–0.35	–0.05
0.2	–0.1
0.35	–0.15
0.35	–0.2

If you make a mistake, select Del Points from the Edit menu and start over. Every time you create a point, the location of it is saved into a buffer in memory. You can execute other commands, like changing views, then continue creating other points. As long as the Del Points command is not used, they remain in memory (most of the commands in the Path menu also erase the point buffer).

Once the points are created, there are two ways to create a Sweep object: either by sweeping the current points, or by saving the points to a file and then sweeping them. We will use the second (file) method on the Sweep object and the first (points) method on the Extrude object. Saving the data is convenient since there are various commands that can read data files and either display them like the Load command inside of the Path menu or the B-Spline Data File also inside the Path menu. The B-Spline command smooths out a data file and saves it to a different file. This way, the original data is not lost.

7. Select the Path menu.
8. Pick the Freehand command.
9. Save the data as VASE.DAT.

Initially, the Path menu was developed to aid in the creation of animation files. It is convenient to create data files that would represent different locations of the viewpoint or an object. These could then be used to generate a series of frames that could be merged together later to produce an animation. Interestingly enough, you can use data files (or path files, as POVCAD refers to them) to produce curves and contours in Sweep or Extrude objects. Also, they are very handy when creating letters, as you see later in the chapter. The Path menu consists of the following commands



Path Menu:

Load Loads a data file into POVCAD

Save Applies transformations to current data files and saves as *.PTH

Freehand Saves the point buffer to a data file

Linear Divides a line into sections and saves to a data file

Circular Creates an arc with line segments and saves to a data file

Parabolic Creates a parabola with line segments and saves to a data file

Helix Creates a helix with line segments and saves to a data file

B-Spline Smooths the point buffer data and saves to a file

Merge Outer/Inner Given an outer and an inner curve, converts both into an outer curve and saves to a data file

Join 2 Curves Patches two data files together and saves to a data file

Flip Data File Rotates the data so the beginning point becomes the end point and vice-versa

B-Spline Data File Smooths the data from a file and saves to a data file

A data or path file is basically a file that contains a list of points that define a curve. Steps 8 and 9 take the points in the buffer and save them to a file.

Let's now create the Sweep object:

1. From the Object menu, select Sweep and then Data File.
2. A file selection box will appear. Pick the file VASE.DAT. When prompted for a name, use VASESWP.RAW.

   The Sweep command creates flat triangle information that is saved as a RAW data file. You do not have to use the *.RAW extension, but it becomes a good reference later.
3. The sweep angle is 360 degrees and the # of segments is 12.
4. Click the Iso View (see fig. 7.14).

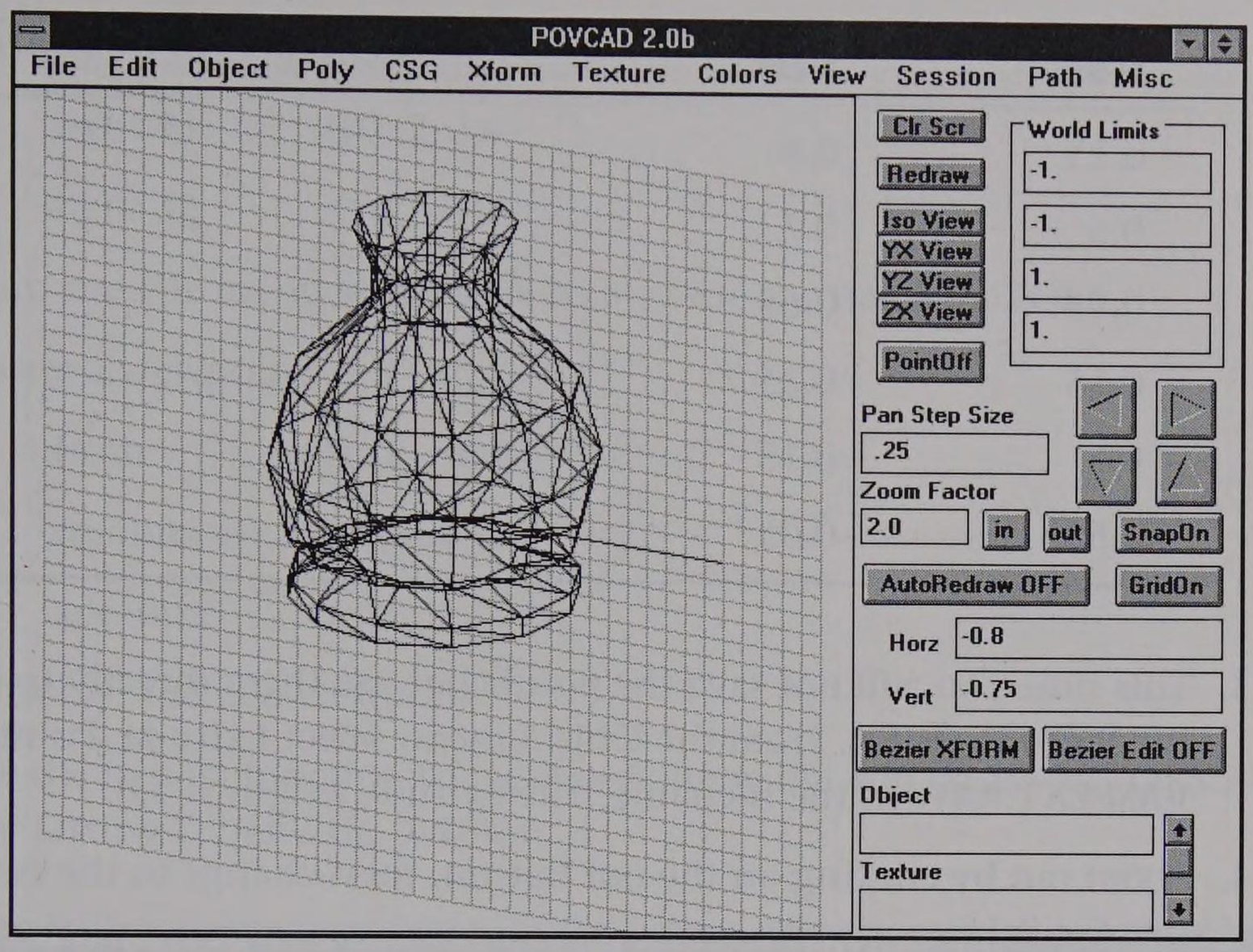


Figure 7.14

Sweep object created with VASE.DAT.

5. Click Session, select the first object and click OK.
6. Move the mouse to the drawing area and press the right mouse button. When the window appears, select Delete. Answer YES.

   This step has deleted the VASE.DAT file from memory but not from the hard drive.
7. Click Redraw to refresh the screen.
8. Select the YX View.
9. Click Session and select the SWP object (Sweep) and click OK.
10. Xform:

Trans	x	–0.8
Trans	y	0.2

11. Select Texture, white_marble.

You have completed the creation of the Sweep object. Our next step is to create the Extrude object:

1. Click the PointOff button to turn on points creation.
2. Create the following points on the screen:

Horz	Vert
0.15	0.8
0.5	0.7
0.15	0.65
0.15	0.25
0.5	0.15
0.55	–0.05

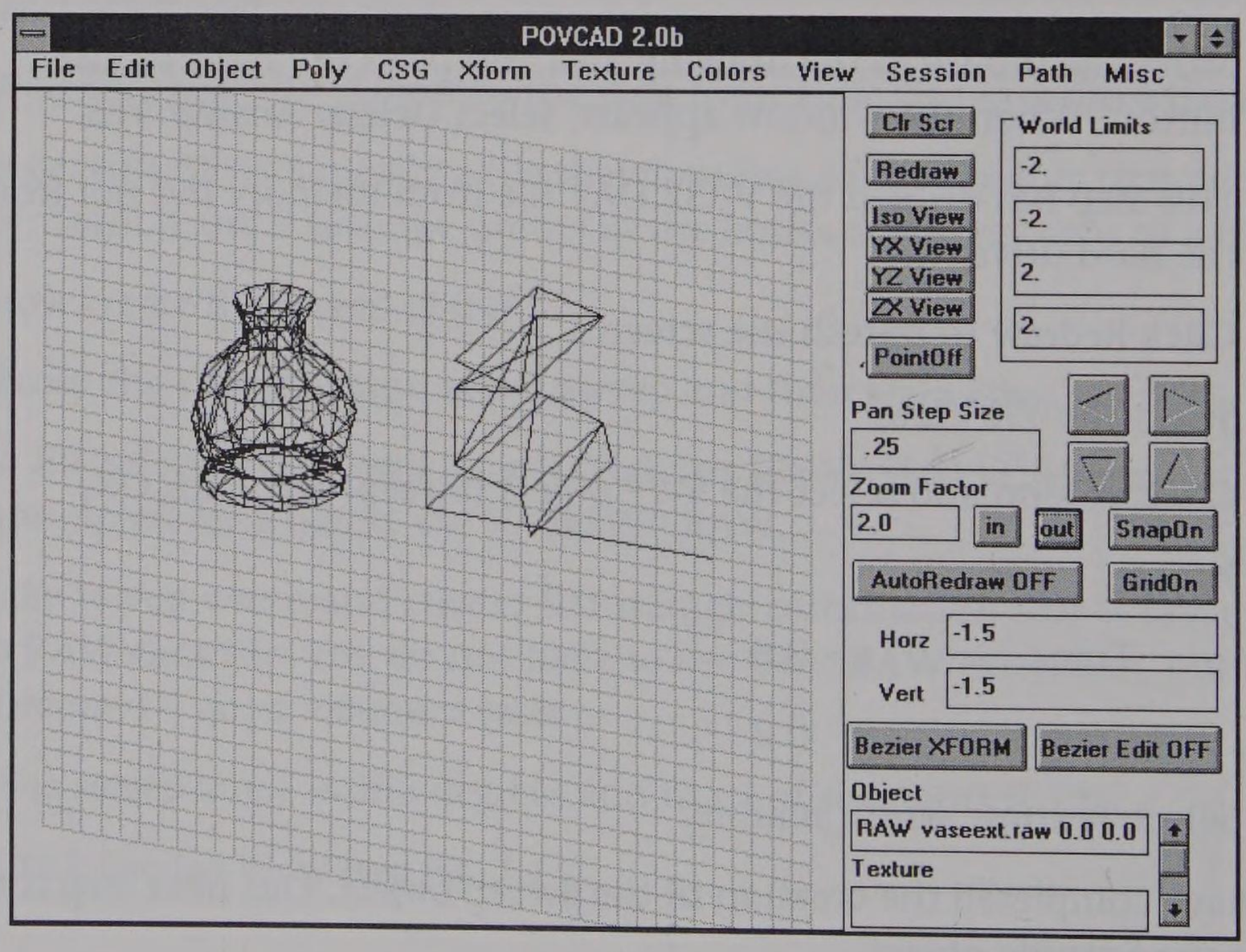
3. This time you will not save the points but use them directly to create the Extrude object. From the Object menu select Extrude, Points; use VASEEXT.RAW as the filename. The extrude length is 1.0.

4. Zoom out by clicking on the out button, then change to the Iso View (see fig. 7.15).

5. Select Texture, reflective_red.

Figure 7.15

Sweep (left) and Extrude (right) objects.



6. Create two light sources:

   light #1 <10, 2, 1>

   light #2 <0, 2, –10>

7. From the File menu select Save As and use the name TUTOR5.

8. Create a viewpoint:

   from <1, 1, –4>

   resolution 100,100

   aperture 0

   Click on "include viewpoint."

9. Use the Export function to create a Polyray file and create TUTOR5.PI (see Listing 7.7 and fig. 7.16).

Listing 7.7. Sweep and extrude object tutorial (TUTOR5.PI).

(Note: Listing was truncated due to its large size.)

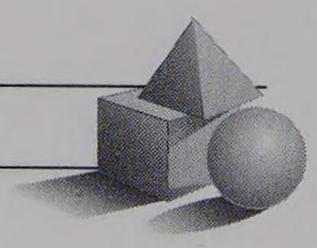
```
//PROGRAM:   TUTOR5.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/3/93 7:32:55 PM

include "colors.inc"
include "texture.inc"

viewpoint {
 from      <1, 1, -2.0 >
 at        <-0.1, 0.4, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 hither     1.0e-3
 resolution 100,100
 aspect     1.0
 yon        1.0e5
 max_trace_depth 5
 aperture        0
}

// RAW DATA
object{
 object {
   polygon 3,
    <0.15,0.65,0.0>,
    <0.2,0.75,0.0>,
    <0.1732,0.75,-0.1>
 }
+
```

continues

Listing 7.7. Continued.

```
 object {
   polygon 3,
    <0.1299,0.65,-0.075>,
    <0.15,0.65,0.0>,
    <0.1732,0.75,-0.1>
 }

//  This section was deleted to reduce the size of this listing

+
 object {
   polygon 3,
    <0.35,-0.2,0.0>,
    <0.3031,-0.2,0.175>,
    <0.35,-0.15,0.0>
 }
    translate < -0.8,0.2,0.0>
   white_marble
}


// RAW DATA
object{
 object {
   polygon 3,
    <0.5,0.7,0.0>,
    <0.15,0.8,0.0>,
    <0.15,0.8,1.0>
 }
+

//  This section was deleted to reduce the size of this listing

 object {
   polygon 3,
    <0.55,-0.05,1.0>,
    <0.55,-0.05,0.0>,
    <0.5,0.15,1.0>
 }
   reflective_red
}
// LIGHT_SOURCE
  light < 10 ,  2 ,  1  >

// LIGHT_SOURCE
  light < 0 ,  2 , -10  >
```

The complete TUTOR5.PI is roughly 32 pages long! Keep the length in mind when you create the same scene using Polyray's Lathe and Sweep object definitions.



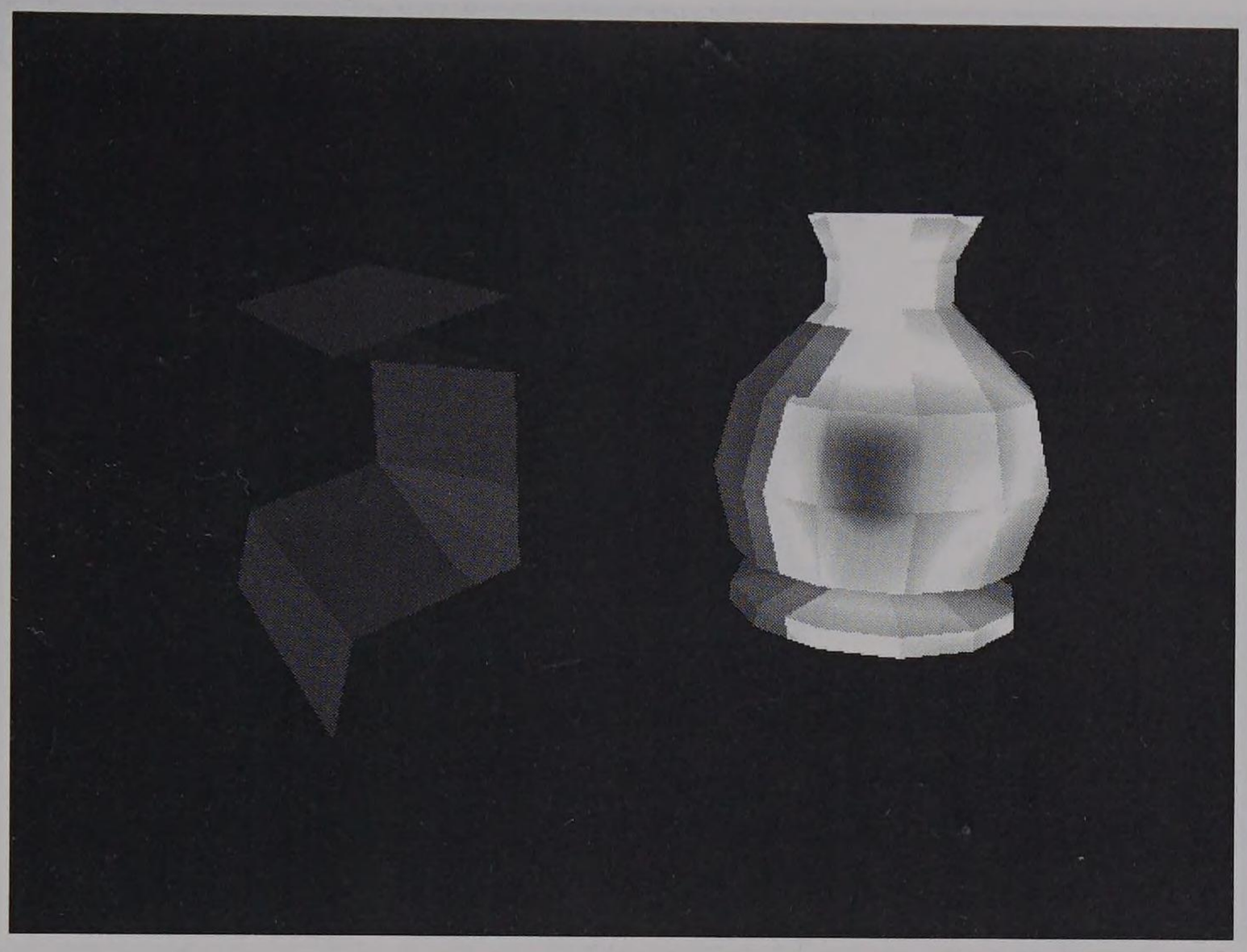


Figure 7.16

Ray-traced image of TUTOR5.PI created with POVCAD.

In order to create a Lathe object for Polyray, I used the data file VASE.DAT with the program LATHE.BAS, used in Chapter 3 to create Listing 3.16 and figure 3.13. I renamed TUTOR5.PI to TUTOR5M.PI and in it I deleted the original objects and added the Lathe and Sweep objects using Polyray's format (see Listing 7.8).

Listing 7.8. Simplifying TUTOR5.PI (TUTOR5M.PI).

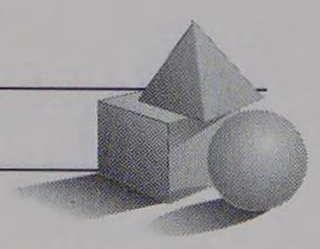
```
//TUTOR5M.PI
//Modified by Hand
//Lathe Object Created With LATHE.BAS
//08-03-1993 19:54:30

include "colors.inc"
include "texture.inc"


define LatheTexture white_marble
define ExtrudeTexture reflective_red

viewpoint {
 from      <1, 1, -2.0 >
 at        <-0.1, 0.4, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
```

continues

Listing 7.8. Continued.

```
 hither      1.0e-3
 resolution 100,100
 aspect      1.0
 yon         1.0e5
 max_trace_depth 5
 aperture         0
}


object {
    lathe  2 , <0, 1, 0>,  9 ,
           <    0.200,    0.750,    0.000>,
           <    0.150,    0.650,    0.000>,
           <    0.150,    0.550,    0.000>,
           <    0.350,    0.350,    0.000>,
           <    0.400,    0.150,    0.000>,
           <    0.350,   -0.050,    0.000>,
           <    0.200,   -0.100,    0.000>,
           <    0.350,   -0.150,    0.000>,
           <    0.350,   -0.200,    0.000>
    translate <-0.8, 0.2, 0.0>
    LatheTexture
}

object {
    sweep  2 , <0, 0, 1>,  6 ,
           <    0.15 ,    0.80 ,    0.0  >,
           <    0.50 ,    0.70 ,    0.0  >,
           <    0.15 ,    0.65 ,    0.0  >,
           <    0.15 ,    0.25 ,    0.0  >,
           <    0.50 ,    0.15 ,    0.0  >,
           <    0.55 ,   -0.05 ,    0.0  >
    ExtrudeTexture
}


// LIGHT_SOURCE
  light < 10 ,  2 ,  1  >

// LIGHT_SOURCE
  light < 0 ,  2 , -10  >
```

Listing 7.8 is the complete version! Compare it to POVCAD's 32-page version of the same scene. The file TUTOR5M.PI lets Polyray do all the hard work while the POVCAD version breaks the objects into polygons. Figure 7.17 displays the image generated with Listing 7.8.



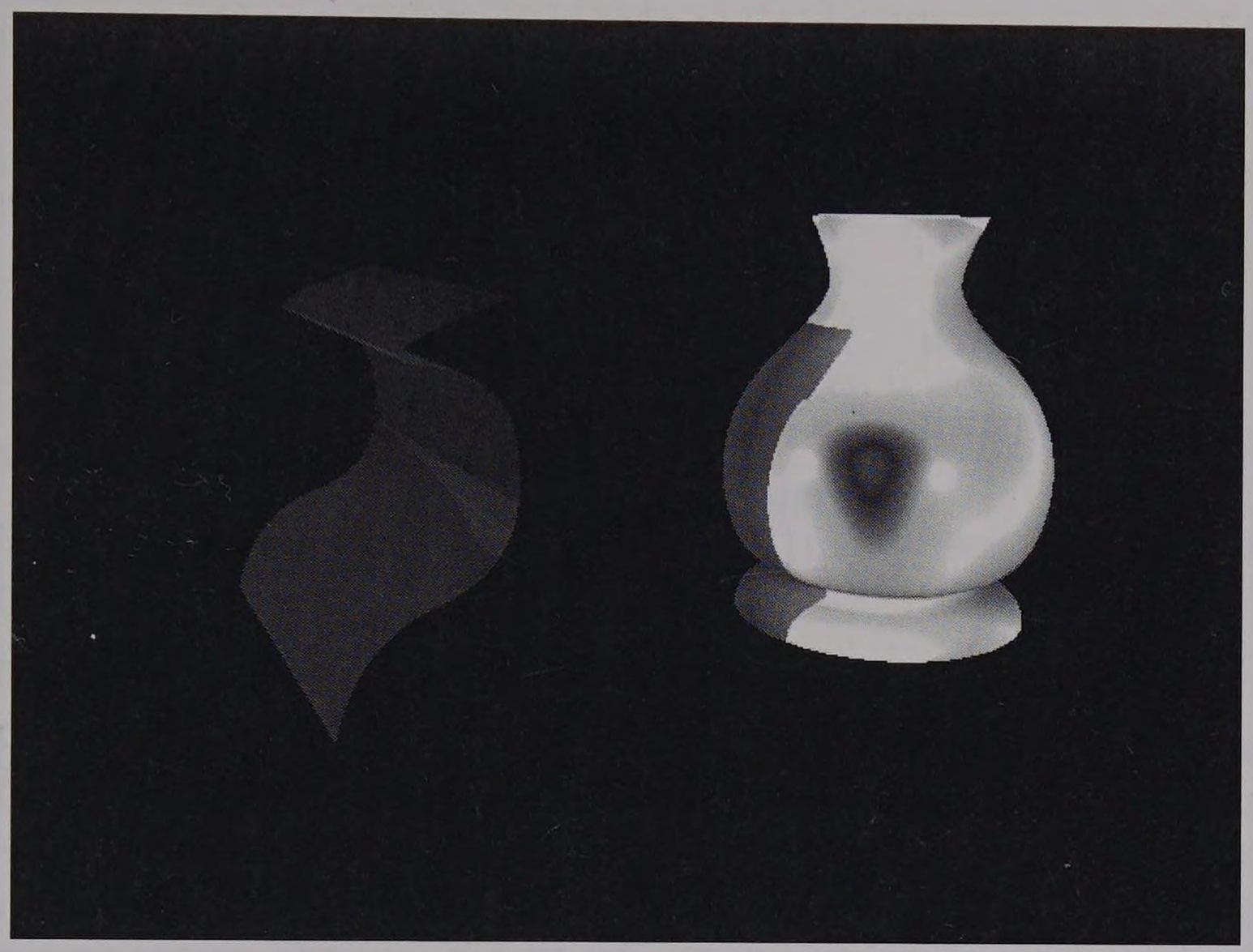


Figure 7.17
Using Polyray's lathe and sweep objects.

In symmetric objects like the vase created in POVCAD, Polyray does a great job. POVCAD has an advantage in that it can read a RAW data file which may not represent a symmetric object or a solid of revolution, such as a RAW data file of a plane or a face. Here, POVCAD can read the file and generate a set of polygons to model it. You may even want to use the program SMOOTH.BAS (from Chapter 3) to create smooth triangles. To read a RAW data file, use the RAW command in the Object menu. Polyray, on the other hand, can't use the Lathe or Sweep object on irregular RAW data files.

Creating Bézier Patches (Tutorial #6)

Being able to control Bézier patches is extremely interesting. POVCAD supports various Bézier patch based objects. These are the commands available under the Object menu, Bézier submenu:

Bézier SubMenu:

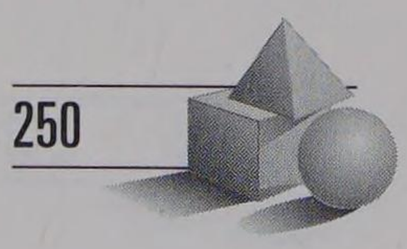
Load A File	Reads files that were created with POVCAD (*.BEZ)
Flat	Flat sheet
Cylinder 2 patch	Cylinder composed with 2 patches
Cylinder 4 patch	Cylinder composed with 4 patches
Ellipsoid	Ellipsoid
Cone	Cone
Torus	Hemi-Torus

In this session, you learn how to deform the Ellipsoid object. POVCAD generates a control mesh that defines the boundaries of the patches. The control mesh has nodes you can select and translate on any axis. The technique is to use the Iso View to select the control node and then change to a different view to displace it. The Iso View is good for picking a point from the screen but not for displacing them. The other views are ideal for doing translations on one plane at a time. Let's get started with the Bézier patches:

1. Select the Iso View.
2. From the Objects menu, select Bézier and then Ellipsoid.

   POVCAD generates a file with the control data. You must give it a name and it is suggested that you use the BEZ extension. Only files ending in BEZ can be later read by the Load A File command in the Bézier menu. Enter the name ELLIP.BEZ.
3. From the Misc menu, select Bézier Setup.

   In this window, you can select to see the control grid, the Bézier patch surface, or neither. In addition, you can specify how many divisions per side to make on the surface. The higher the number, the better the image it produces in POVCAD, but the program runs slower due to the increased amount of redrawing that has to be done. Set the number of divisions at 13, and select both the control grid and the surface patch.
4. Click SnapOff and GridOff.
5. To start deforming the Bézier object, click Bézier Edit Off (this enables editing).



POVCAD draws the control grid in black while the Bézier surface is drawn in blue. To deform the surface, click any of the intersections of the black lines (the nodes). Then click in a new location again to tell POVCAD the new position of the node.

6. Click the mouse at Horz = 0.0 and Vert = 0.5.
7. A red circle appears on the node that has been selected.
8. Select the YX View.
9. Click a point at Horz = 0.5 and Vert = 0.7 (the surface must deform at this point).
10. Click at Horz = 0.0 and Vert = –0.5 (selecting a node).
11. Click at Horz = 0.5 and Vert = –0.8 (deformation).
12. Click at Horz = –0.5 and Vert = 0.0 (selecting).
13. Click at Horz = –0.8 and Vert = 0.0 (deforming).
14. From the Misc menu, select Bézier Setup. Disable the control grid and set the number of divisions to 13 (maximum).
15. Click Redraw to refresh the screen.
16. Change to the Iso View (see fig. 7.18).

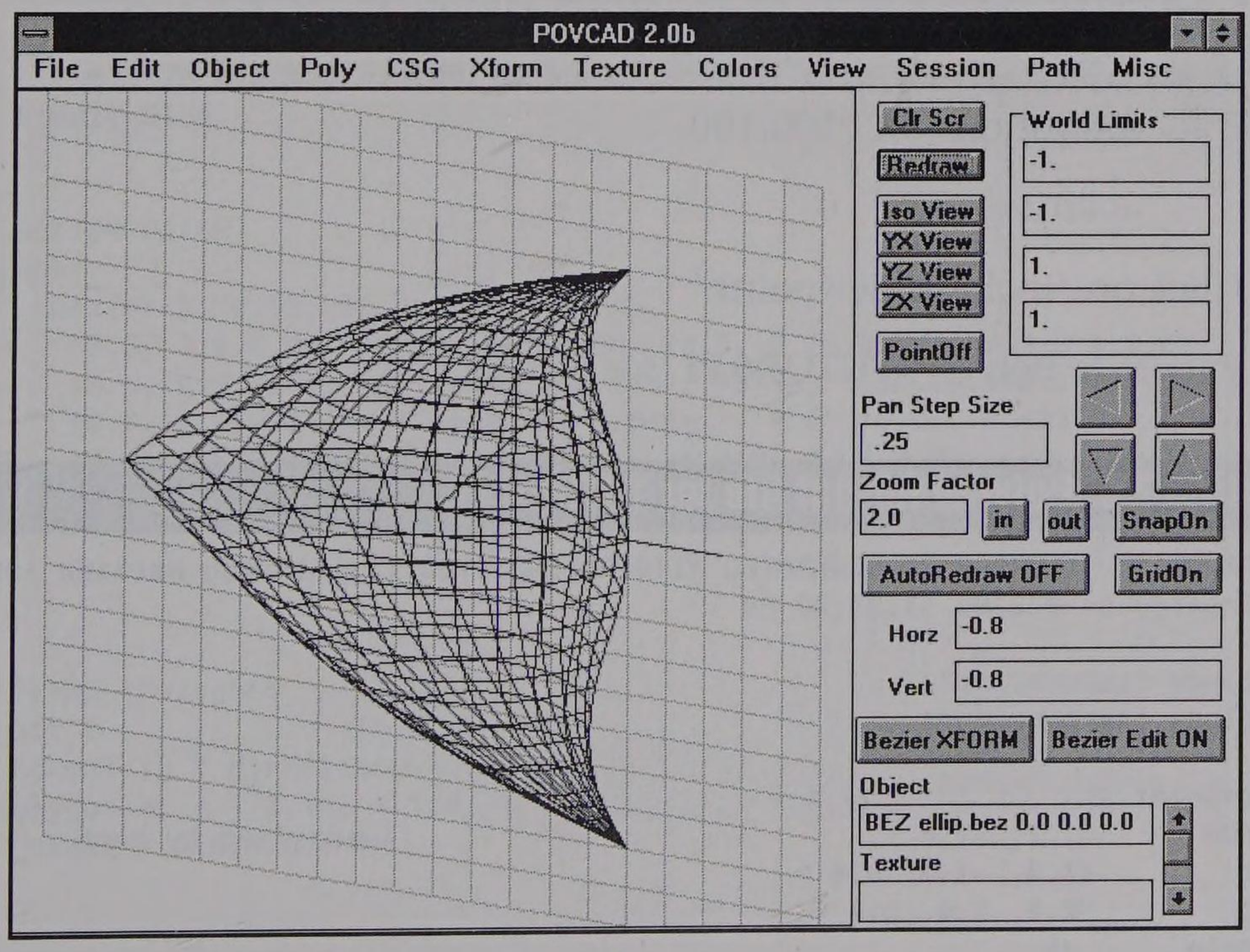


Figure 7.18
Deforming a Bézier patch.

17. Select Texture, steely_blue.
18. From the Object menu select Plane_Y.
19. Xform:

Scale x, y, z	3 (all of them)
Trans y	–1.5

20. Select Texture, mirror.
21. From the Object menu, select Plane_Z.
22. Xform:

Scale x, y, z	3 (all of them)
Trans y	2

23. Select Texture, mirror.
24. Create two light sources:

light #1	<0, 5, –10>
light #2	<5, 3, –10>

25. From the File menu, select Save As: TUTOR6
26. Create a viewpoint:

from	<1, 3, –5>
resolution	100,100
aperture	0

Click on "include viewpoint."

27. Export to Polyray: TUTOR6.PI (see Listing 7.9).

Listing 7.9. Polyray format file for TUTOR6.PI.

```
//PROGRAM:   TUTOR6.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/3/93 11:35:02 PM

include "colors.inc"
include "texture.inc"

viewpoint {
 from     <1, 2, -7 >                                  // modified by hand
 at        <0.0, -1.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
```

```
 hither      1.0e-3
 resolution 100,100
 aspect      1.0
 yon         1.0e5
 max_trace_depth 5
 aperture         0
}

object{
// BEZIER PATCH
object{
     bezier 2, 0.05, 3, 3,
      <0.5,0.7,0.0>, <0.28,0.5,0.0>, <0.5,0.28,0.0>, <0.5,0.0,0.0>,
      <0.5,0.7,0.0>, <0.14,0.5,0.242>, <0.25,0.28,0.433>, <0.25,0.0,0.433>,
      <0.5,0.7,0.0>, <-0.14,0.5,0.242>, <-0.25,0.28,0.433>, <-0.25,0.0,0.433>,
      <0.5,0.7,0.0>, <-0.28,0.5,0.0>, <-0.5,0.28,0.0>, <-0.8,0.0,0.0>
   steely_blue
}
+
// BEZIER PATCH
object{
     bezier 2, 0.05, 3, 3,
      <0.5,0.7,0.0>, <0.28,0.5,0.0>, <0.5,0.28,0.0>, <0.5,0.0,0.0>,
      <0.5,0.7,0.0>, <0.14,0.5,-0.242>, <0.25,0.28,-0.433>, <0.25,0.0,-0.433>,
      <0.5,0.7,0.0>, <-0.14,0.5,-0.242>, <-0.25,0.28,-0.433>, <-0.25,0.0,-0.433>,
      <0.5,0.7,0.0>, <-0.28,0.5,0.0>, <-0.5,0.28,0.0>, <-0.8,0.0,0.0>
   steely_blue
}
+
// BEZIER PATCH
object{
     bezier 2, 0.05, 3, 3,
      <0.5,-0.8,0.0>, <0.28,-0.5,0.0>, <0.5,-0.28,0.0>, <0.5,0.0,0.0>,
      <0.5,-0.8,0.0>, <0.14,-0.5,0.242>, <0.25,-0.28,0.433>, <0.25,0.0,0.433>,
      <0.5,-0.8,0.0>, <-0.14,-0.5,0.242>, <-0.25,-0.28,0.433>, <-0.25,0.0,0.433>,
      <0.5,-0.8,0.0>, <-0.28,-0.5,0.0>, <-0.5,-0.28,0.0>, <-0.8,0.0,0.0>
   steely_blue
}
+
// BEZIER PATCH
object{
     bezier 2, 0.05, 3, 3,
      <0.5,-0.8,0.0>, <0.28,-0.5,0.0>, <0.5,-0.28,0.0>, <0.5,0.0,0.0>,
      <0.5,-0.8,0.0>, <0.14,-0.5,-0.242>, <0.25,-0.28,-0.433>, <0.25,0.0,-0.433>,
      <0.5,-0.8,0.0>, <-0.14,-0.5,-0.242>, <-0.25,-0.28,-0.433>, <-0.25,0.0,-0.433>,
      <0.5,-0.8,0.0>, <-0.28,-0.5,0.0>, <-0.5,-0.28,0.0>, <-0.8,0.0,0.0>
   steely_blue
}
}


// PLANE (Polygon)
object {
// normal in Y direction
  polygon 4, <-1,0,1>, <-1,0,-1>, <1,0,-1>, <1,0,1>
    scale < 3.0,3.0,3.0>
```

continues

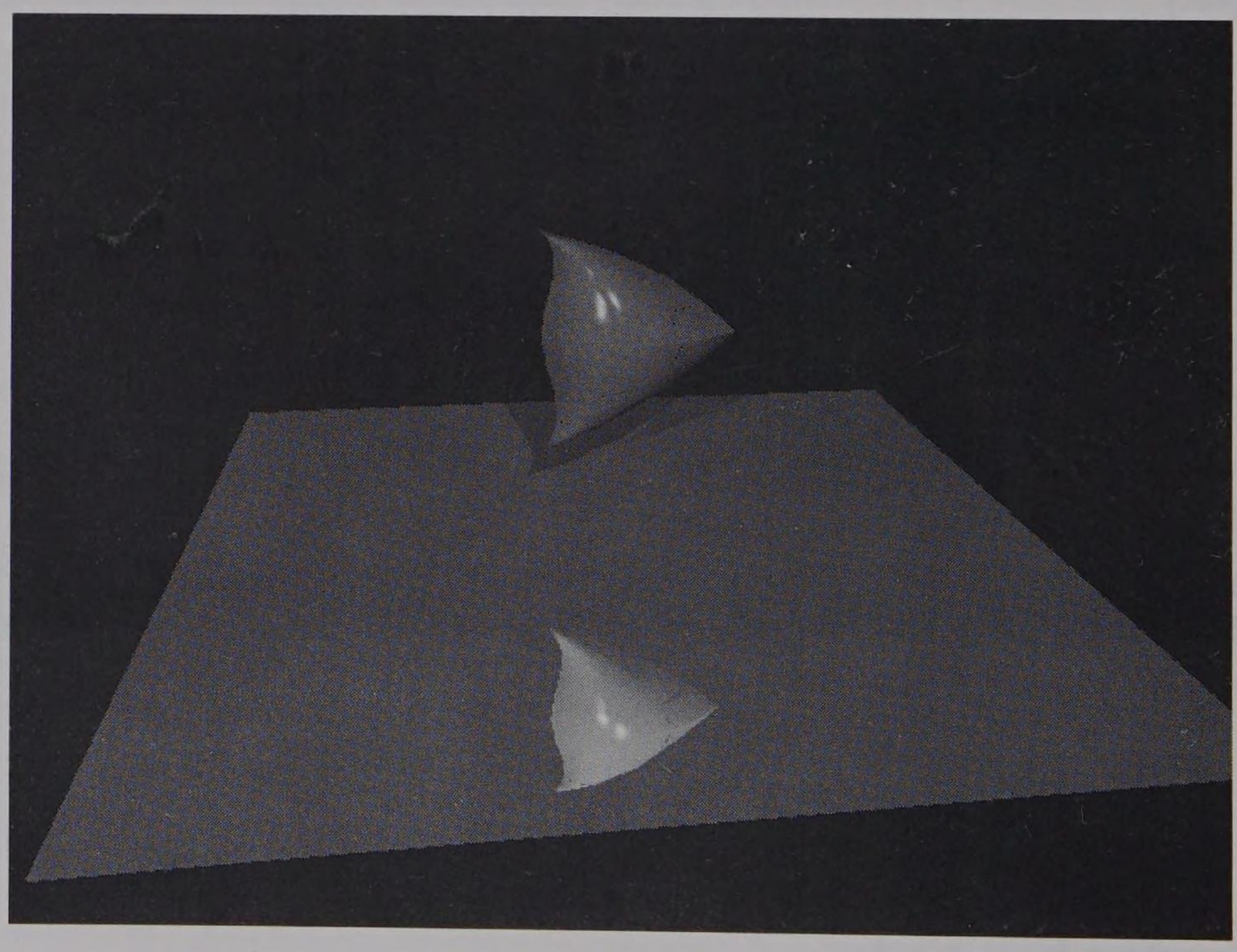
Listing 7.9. Continued.

```
  translate < 0.0,-1.5,0.0>
   mirror
}

//  This plane was commented out by hand

/*
// PLANE (Polygon)
object {
// normal in Z direction
  polygon 4, <-1,1,0>, <1,1,0>, <1,-1,0>, <-1,-1,0>
    scale < 3.0,3.0,3.0>
    translate < 0.0,0.0,2.0>
   mirror
}
*/

// LIGHT_SOURCE
  light < 0 ,  5 , -10  >

// LIGHT_SOURCE
  light < 5 ,  3 , -10  >
```

Listing 7.9 presents the modified version of TUTOR6.PI. One of the planes was commented out and the viewpoint was also reworked (see fig. 7.19).

Figure 7.19

A Bézier patch object.



Using the Polygon Decomposition Technique (Tutorial #7)

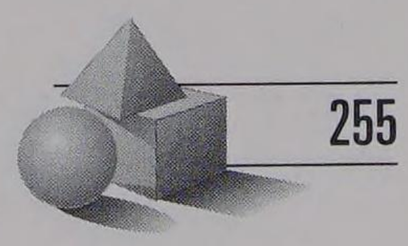
Polygon decomposition refers to the technique used to convert a close curve into a flat surface. The surface will be composed of triangles. In this session, you create a 3-D letter "A" by creating the contours of it. First you create the outer contour of the letter and then the hole in the middle. Save them as data files. Then you extrude the contours. Finally, you use the Poly menu on the outer and inner contours to create a flat surface which will represent the "face" of the letter. Again, we'll compare this method with the use of Polyray's sweep objects.

An important point: The polygon decomposition method always assumes that the curves are created in the YX plane. The routines might work in other planes, but I can't guarantee it. You can always start in the YX plane then Xform to the final location.

Start by creating curves that define the contours of your letter "A":

1. Select the YX View.
2. Enable SnapOn and GridOn.
3. From the View menu, select a Grid Spacing of 40.
4. Enable PointOn.
5. Create the following points on the screen:

Horz	Vert
–0.2	0.5
0.2	0.5
0.55	–0.35
0.25	–0.35
0.15	–0.05
–0.15	–0.05
–0.25	–0.35
–0.55	–0.35
–0.2	0.5



6. From the Path menu, select B-Spline. Input AOUTER.DAT file and input 50 points.

7. From the Edit menu, select Del Points. Enable point creation by clicking on PointOff.

8. Create the following points on the screen:

Horz	Vert
–0.1	0.3
0.05	0.3
0.1	0.1
–0.05	0.15
–0.2	0.1
–0.1	0.3

9. Disable points creation, click on PointOn.

10. From the Path menu select B-Spline. Input AINNER.DAT file and input 18 points. Now erase the points by selecting the Edit menu and clicking on Del Points.

    At this point, both contours should be smooth on the screen after doing a redraw (see fig. 7.20).

    Take the next steps to extrude the curves:

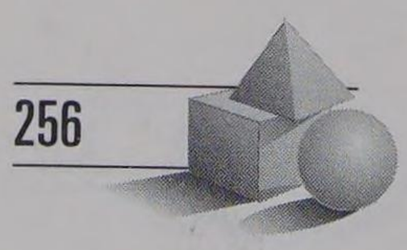
11. From the Object menu select Extrude, Data File. Pick AOUTER.DAT, Save As AOUTER.RAW and use an extrude length of 0.5 units.

12. Select the Iso View.

13. From the Object menu select Extrude, Data File. Pick AINNER.DAT, Save As AINNER.RAW and use an extrude length of 0.5 units (see fig. 7.21).

    The sides and hole of the letter "A" are done. The remaining piece is the "face" of the letter.

14. From the Poly menu select Perforated. The file box appears. Pick the files AOUTER.DAT (which is the outer curve), then when the file box appears again select AINNER.DAT (the inner curve). If you're not sure,



read the title in the file box. It tells you which curve (outer or inner) is needed at that time.

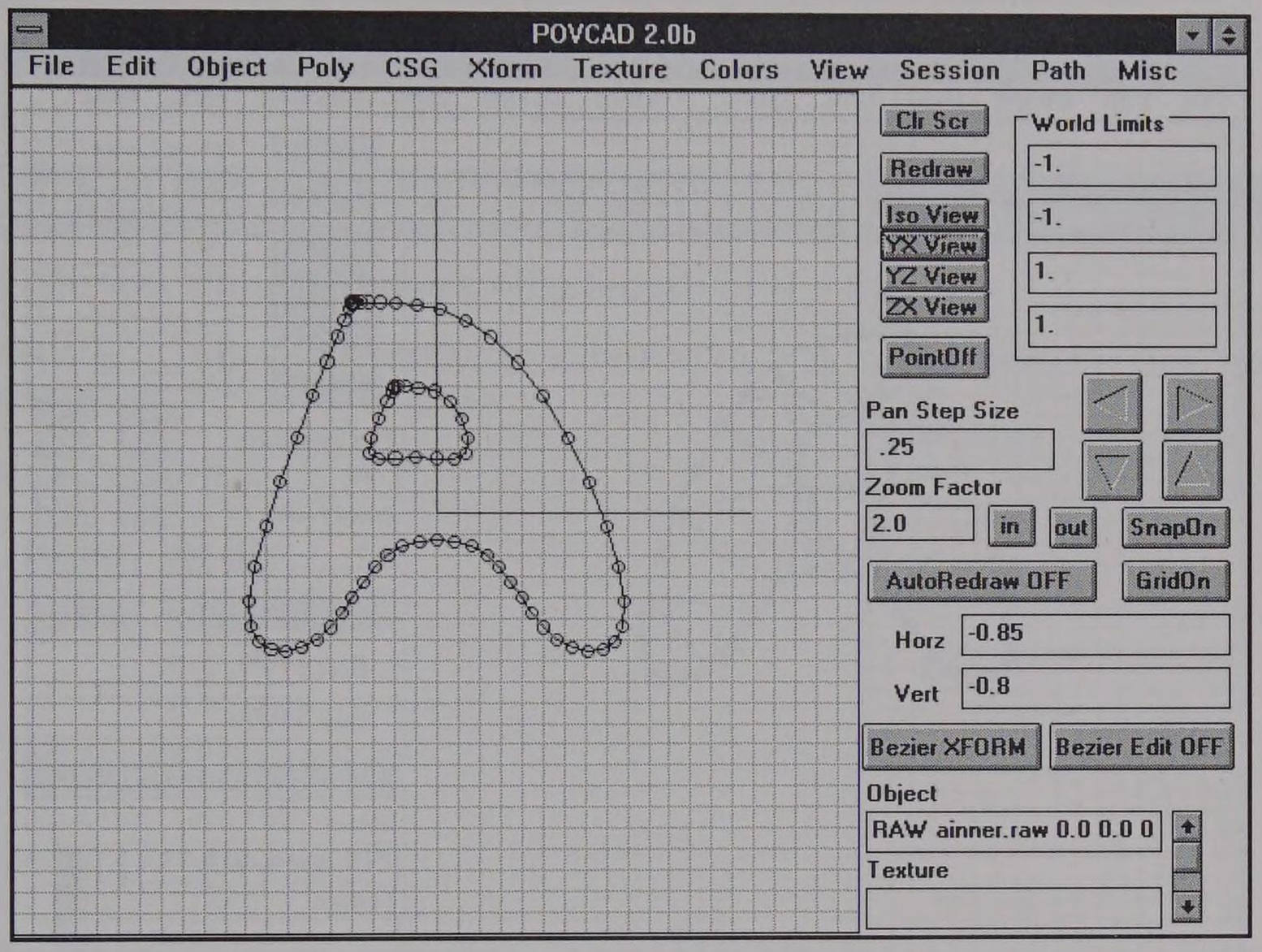


Figure 7.20

Smooth contours of the letter "A."

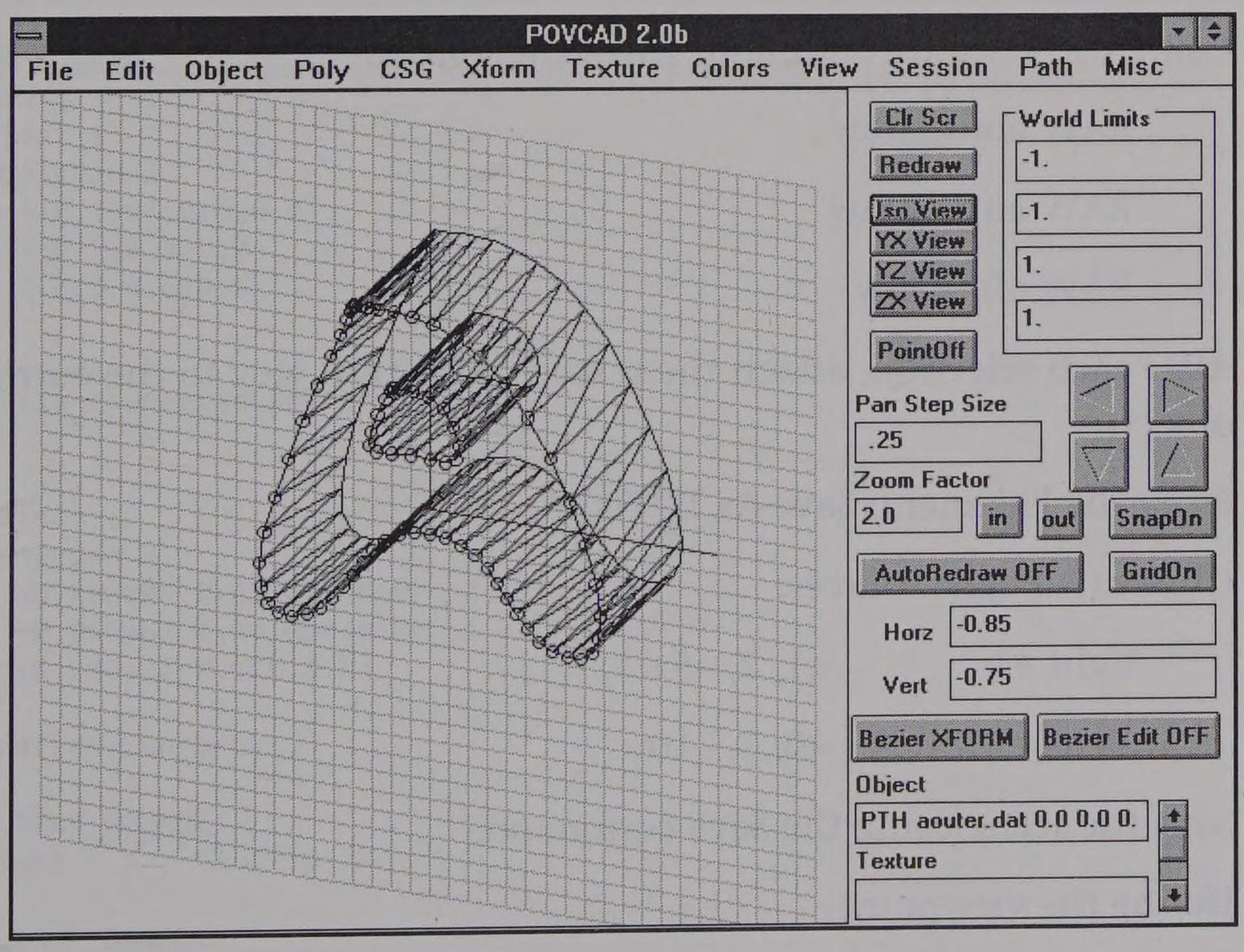


Figure 7.21

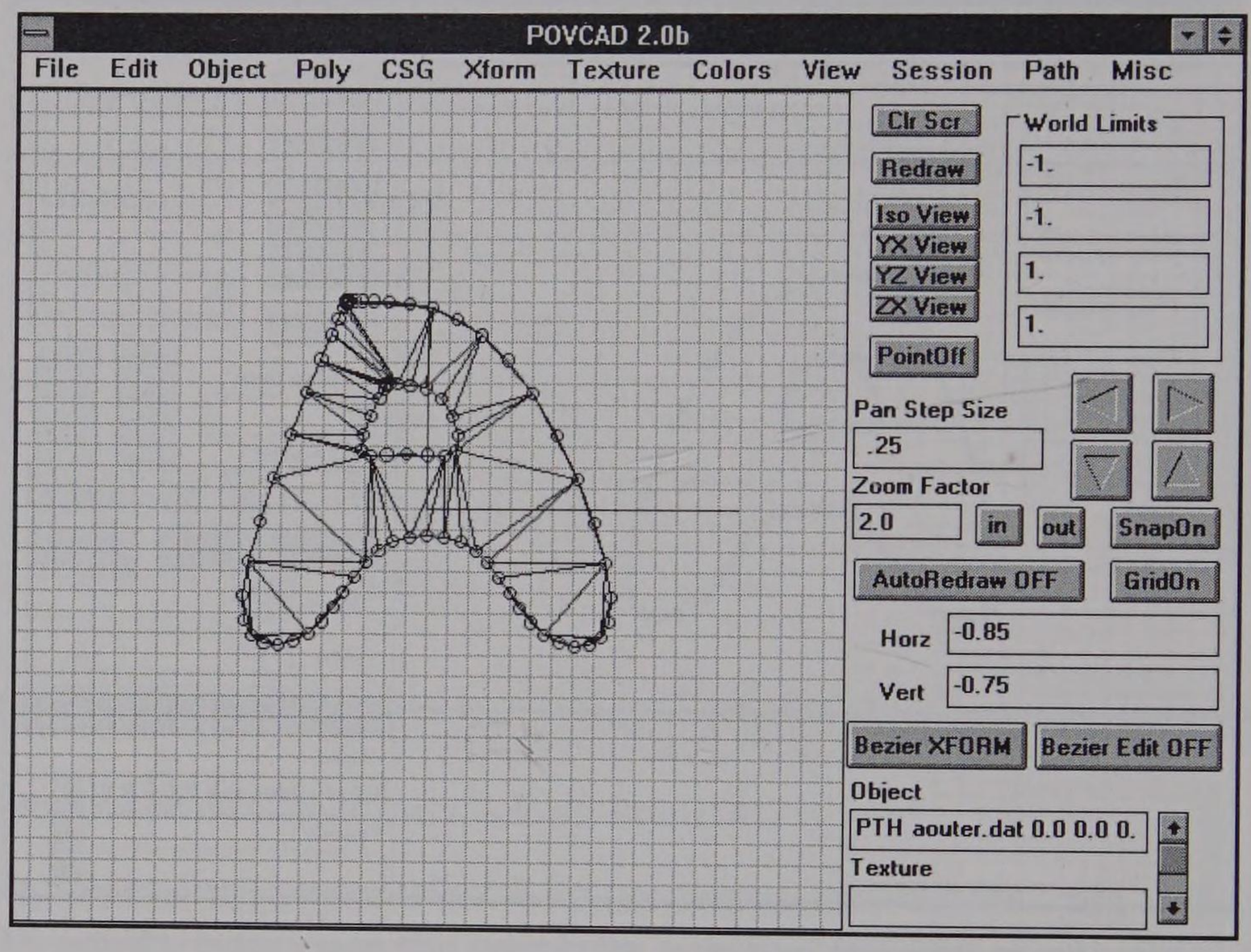
Extrusion of contours.

15. The output filename is AFACE.RAW.

16. Select the YX View (see fig. 7.22).

Figure 7.22

Polygon decomposition applied to face of letter "A."



17. Using the Session command, select the objects

    RAW aouter.raw

    RAW ainner.raw

    RAW aface.raw

    After each selection, attach the texture reflective_grey to the current object.

18. From the File menu Save As TUTOR7.

19. Create two light sources:

light #1	<1, 2, –5>
light #2	<0, 1, –10>

20. Save the scene using the Save command.

21. Define the viewpoint

from	<1, 2, –5>
resolution	100,100

aperture 0

Click on "include viewpoint."

22. To complete the process, Export to Polyray as TUTOR7.PI (see Listing 7.10 and fig. 7.23).

Listing 7.10. Creating the letter "A" (TUTOR7.PI).

(Note: file was truncated due to the length of the file)

```
//PROGRAM:    TUTOR7.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/4/93 1:25:07 AM

include "colors.inc"
include "texture.inc"

viewpoint {
 from       <1, 1, -1.4 >
 at         <0.2, 0.2, 0.0 >
 up         <0.0, 1.0, 0.0 >
 angle       45
 hither      1.0e-3
 resolution 100,100
 aspect      1.0
 yon         1.0e5
 max_trace_depth 5
 aperture         0
}

// RAW DATA
object{
 object {
   polygon 3,
    <-0.1997,0.5,0.0>,
    <-0.2,0.5,0.0>,
    <-0.2,0.5,0.5>
 }

// File truncated due to length


+ object {
   polygon 3,
    <0.253009,0.277006,0.0>,
    <0.066667,0.141667,0.0>,
    <0.360417,0.075,0.0>
 }
   reflective_grey
}
// LIGHT_SOURCE
  light < 1 ,  2 , -5  >

// LIGHT_SOURCE
  light < 0 ,  1 , -10  >

// LIGHT_SOURCE
  light < 0 ,  3 , 5  >
```

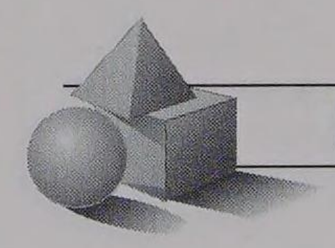
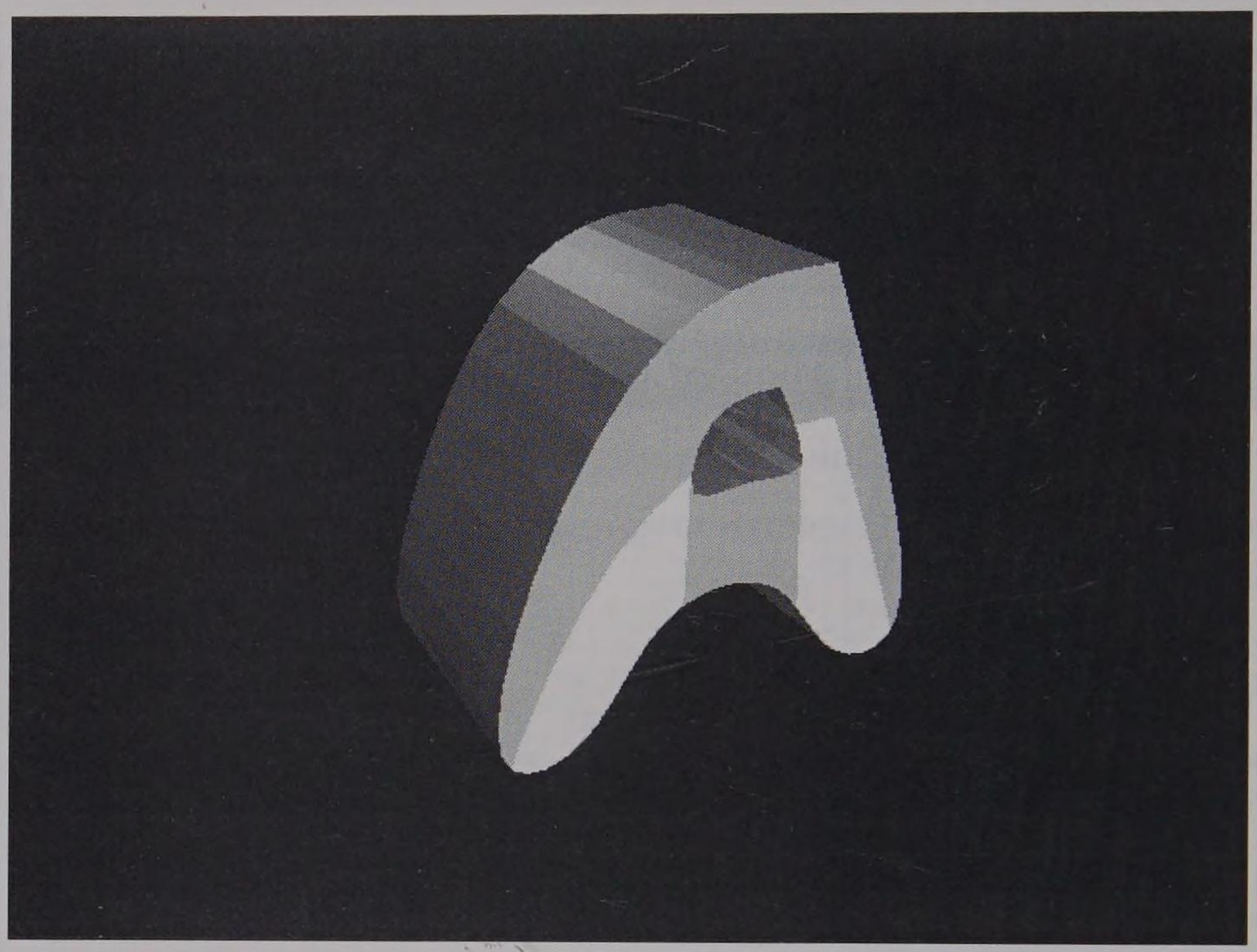


Figure 7.23

Ray-traced letter "A" generated with POVCAD.



Listing 7.10 was extremely long and had to be shortened. In order to compare the geometry created with POVCAD and Polyray's features, I decided to make AOUTER.DAT and AINNER.DAT sweep objects. This shortens the file and creates a smoother surface than using flat triangles. Listing 7.11 presents a modified version of TUTOR7.PI named TUTOR7M.PI and figure 7.24 displays the resulting image of TUTOR7M.PI.

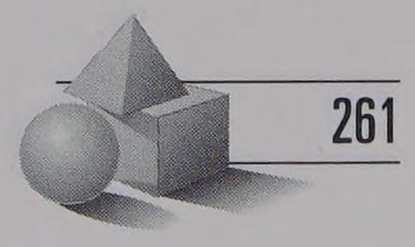
Listing 7.11. Using sweep objects in Polyray.

(Note: part of the file was truncated due to the length.)

```
//PROGRAM:   TUTOR7.PI GENERATED WITH POVCAD 2.0b (c) Alfonso Hermida 1993
//Created on 8/4/93 1:25:07 AM

include "colors.inc"
include "texture.inc"

viewpoint {
 from       <1, 1, -1.4 >
 at         <0.2, 0.2, 0.0 >
 up         <0.0, 1.0, 0.0 >
 angle       45
 hither      1.0e-3
 resolution 100,100
 aspect      1.0
 yon         1.0e5
```

```
 max_trace_depth 5
 aperture        0
}


object {
    sweep  2 , <0, 0, 0.5>,  61 ,
           <   -0.200,    0.500,    0.000>,
           <   -0.200,    0.500,    0.000>,
           <   -0.198,    0.500,    0.000>,
           <   -0.192,    0.500,    0.000>,
           <   -0.180,    0.500,    0.000>,
           <   -0.161,    0.500,    0.000>,
           <   -0.133,    0.500,    0.000>,
           <   -0.095,    0.499,    0.000>,
           <   -0.047,    0.495,    0.000>,
           <    0.007,    0.482,    0.000>,
           <    0.067,    0.458,    0.000>,
           <    0.129,    0.418,    0.000>,
           <    0.192,    0.358,    0.000>,
           <    0.253,    0.277,    0.000>,
           <    0.310,    0.180,    0.000>,
           <    0.360,    0.075,    0.000>,
           <    0.401,   -0.030,    0.000>,
           <    0.429,   -0.127,    0.000>,
           <    0.442,   -0.208,    0.000>,
           <    0.437,   -0.268,    0.000>,
           <    0.419,   -0.306,    0.000>,
           <    0.391,   -0.326,    0.000>,
           <    0.356,   -0.330,    0.000>,
           <    0.319,   -0.320,    0.000>,
           <    0.283,   -0.300,    0.000>,
           <    0.252,   -0.271,    0.000>,
           <    0.225,   -0.237,    0.000>,
           <    0.200,   -0.200,    0.000>,
           <    0.175,   -0.163,    0.000>,
           <    0.148,   -0.129,    0.000>,
           <    0.117,   -0.100,    0.000>,
           <    0.081,   -0.079,    0.000>,
           <    0.041,   -0.067,    0.000>,
           <    0.000,   -0.062,    0.000>,
           <   -0.041,   -0.067,    0.000>,
           <   -0.081,   -0.079,    0.000>,
           <   -0.117,   -0.100,    0.000>,
           <   -0.148,   -0.129,    0.000>,
           <   -0.175,   -0.163,    0.000>,
           <   -0.200,   -0.200,    0.000>,
           <   -0.225,   -0.237,    0.000>,
           <   -0.252,   -0.271,    0.000>,
           <   -0.283,   -0.300,    0.000>,
           <   -0.319,   -0.320,    0.000>,
           <   -0.356,   -0.330,    0.000>,
           <   -0.391,   -0.326,    0.000>,
           <   -0.419,   -0.306,    0.000>,
           <   -0.437,   -0.268,    0.000>,
           <   -0.442,   -0.208,    0.000>,
```

continues



Listing 7.11. Continued.

```
                <    -0.429,    -0.127,    0.000>,
                <    -0.403,    -0.030,    0.000>,
                <    -0.369,     0.075,    0.000>,
                <    -0.330,     0.180,    0.000>,
                <    -0.292,     0.277,    0.000>,
                <    -0.258,     0.358,    0.000>,
                <    -0.234,     0.418,    0.000>,
                <    -0.217,     0.458,    0.000>,
                <    -0.207,     0.482,    0.000>,
                <    -0.202,     0.495,    0.000>,
                <    -0.200,     0.499,    0.000>,
                <    -0.200,     0.500,    0.000>
        reflective_grey
}

object {
    sweep  2 , <0, 0, 0.5>,  22 ,
                <    -0.100,     0.300,    0.000>,
                <    -0.099,     0.300,    0.000>,
                <    -0.093,     0.300,    0.000>,
                <    -0.075,     0.300,    0.000>,
                <    -0.043,     0.299,    0.000>,
                <    -0.004,     0.290,    0.000>,
                <     0.033,     0.267,    0.000>,
                <     0.060,     0.225,    0.000>,
                <     0.073,     0.178,    0.000>,
                <     0.067,     0.142,    0.000>,
                <     0.040,     0.128,    0.000>,
                <    -0.001,     0.130,    0.000>,
                <    -0.050,     0.133,    0.000>,
                <    -0.098,     0.130,    0.000>,
                <    -0.138,     0.128,    0.000>,
                <    -0.158,     0.142,    0.000>,
                <    -0.155,     0.178,    0.000>,
                <    -0.137,     0.225,    0.000>,
                <    -0.117,     0.267,    0.000>,
                <    -0.105,     0.290,    0.000>,
                <    -0.101,     0.299,    0.000>,
                <    -0.100,     0.300,    0.000>
    reflective_grey
}


// RAW DATA
object{
 object {
   polygon 3,
    <-0.094792,0.499344,0.0>,
    <-0.133333,0.5,0.0>,
    <-0.16142,0.5,0.0>
 }

// FACE of Letter A data was truncated to reduce the size of the listing.
```

```
+
 object {
   polygon 3,
    <0.253009,0.277006,0.0>,
    <0.066667,0.141667,0.0>,
    <0.360417,0.075,0.0>
 }
   reflective_grey
}
// LIGHT_SOURCE
  light < 1 ,  2 , -5  >

// LIGHT_SOURCE
  light < 0 ,  1 , -10  >

// LIGHT_SOURCE
  light < 0 ,  3 , 5  >
```

In TUTOR7M.PI, both the outer and inner contours were converted into Polyray's sweep objects. The smoothness of the outer surface and hole can be seen in figure 7.24.

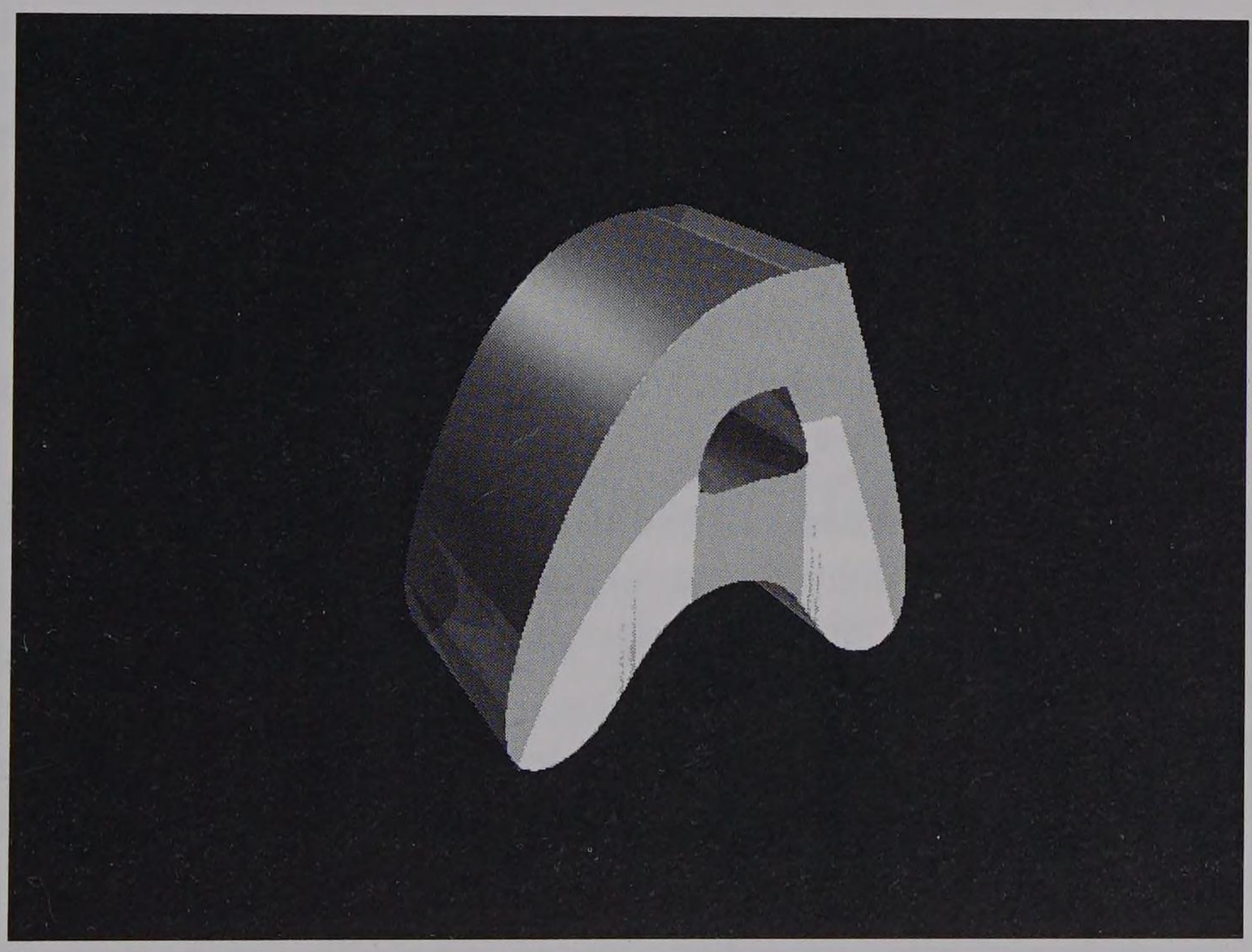


Figure 7.24

Using Polyray's sweep object definition.

Summary

This chapter presented you with a series of seven tutorials demonstrating all the major features of POVCAD. POVCAD is a 3-D wireframe-based modeler

that can help you create scenes for the Polyray ray tracer. After you create the scene in POVCAD, you can generate Polyray-compatible files.

The first tutorial presented a simple scene with three spheres. You learned how to create simple objects and position them in the coordinate system. In addition, you attached textures to each primitive and created light sources. Finally, you prepared a viewpoint definition and generated a Polyray compatible scene file using the Export command.

In Tutorial #2, you saw some of the different primitives available: the cone, torus, box, sphere, and plane.

Tutorial #3 covered the technique of creating multiple copies of objects using the Copy/Rotate command.

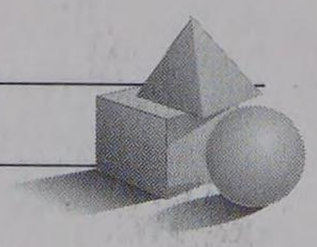
Tutorial #4 dealt with CSG objects. You saw the creation of a primitive Elbow, consisting of the intersection of a torus and a box. The resulting object was a union with two cylinders forming a tube-like shape, bent 90 degrees.

Tutorial #5 was dedicated to POVCAD's Sweep and Extrude objects, called Lathe and Extrude/Sweep in Polyray. You saw these objects compared to equivalent definitions using the Lathe and Sweep objects in Polyray. The quality of the surface using Polyray's objects was outstanding.

In Tutorial #6, you studied the creation and deformation of Bézier patches. You deformed an ellipsoid made with Bézier patches. POVCAD has the option of drawing the control grid, the generated surface, and changing the number of divisions in the surface.

Finally, Tutorial #7 presented polygon decomposition, a technique to create surfaces by subdividing them into triangles. Again, you saw the results compared with Polyray's, and the surface finish using Polyray's objects was better.

In the next chapter, you'll learn how to create animation using Polyray's built-in features. Various techniques will be covered that will discuss how to plan and model an animation with the help of POVCAD.

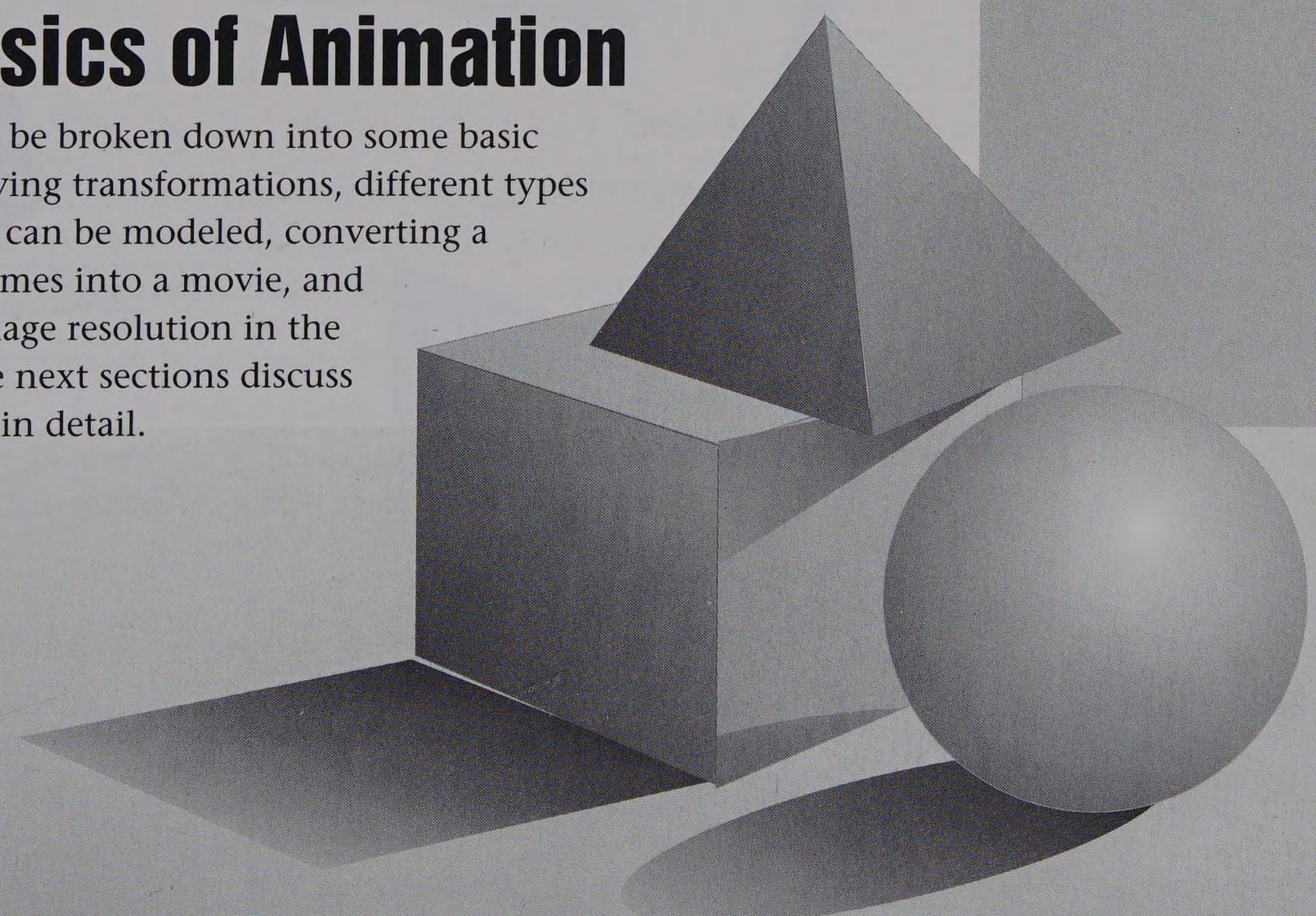


Animation

You are now at the point where you can take ray tracing a step further. The previous chapters were dedicated to understanding the different tools available in Polyray. The scenes used in the examples were all still images. The next step is to combine multiple still frames into an animated sequence or movie. Polyray's ray tracing features, combined with its built-in animation support, make the program ideal for modeling everything from bouncing spheres and orbiting planets to robots and cartoon figures. Let's start with the basics of animation, then cover Polyray's animation features, and finally show various examples of animated sequences, or animations.

The Basics of Animation

Animation can be broken down into some basic concepts: applying transformations, different types of motion that can be modeled, converting a sequence of frames into a movie, and the effect of image resolution in the animation. The next sections discuss these concepts in detail.

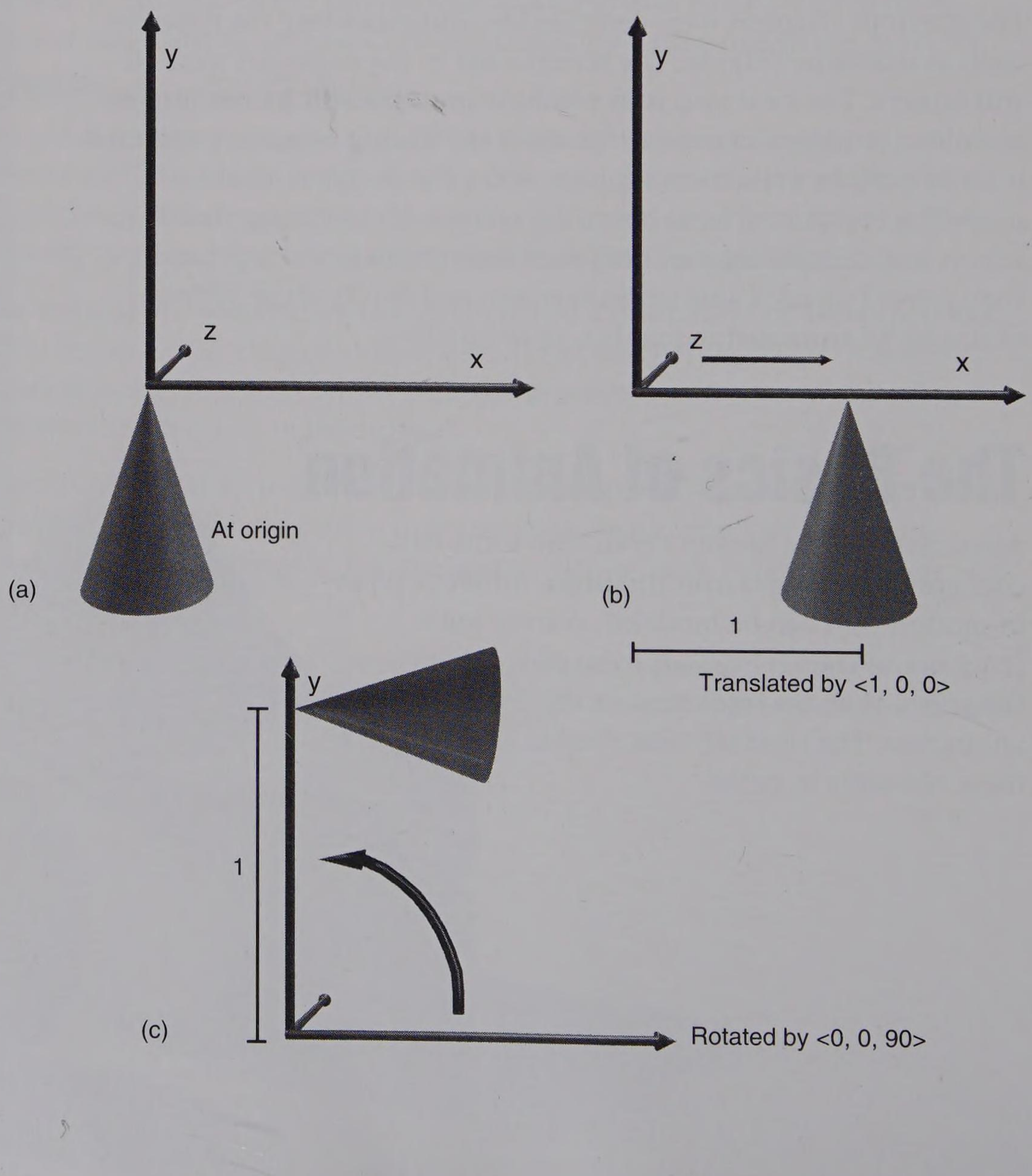


Transformation Operations

Transformations such as scaling, rotation, and translation are the key to object animation. In the previous chapters you used transformations to position objects, create multiple copies, or resize them. In animation, you have to be aware of how the order of the transformations can affect the position and orientation of an object. As an example, take a cone and position it at the origin (see fig. 8.1a). Applying a translation of <1, 0, 0> gives you figure 8.1b. Finally, apply the rotation <0, 0, 90> (see fig. 8.1c). Now, compare this result with the same operations in different order (see fig. 8.2a,b and c). Obviously, the results are not the same. In order to create figure 8.1c using a rotation and then a translation, you would have to apply the rotation <0, 0, 90> and then the translation <0, 1, 0>. This gives you the same result.

Figure 8.1

Applying a translation and a rotation to an object.



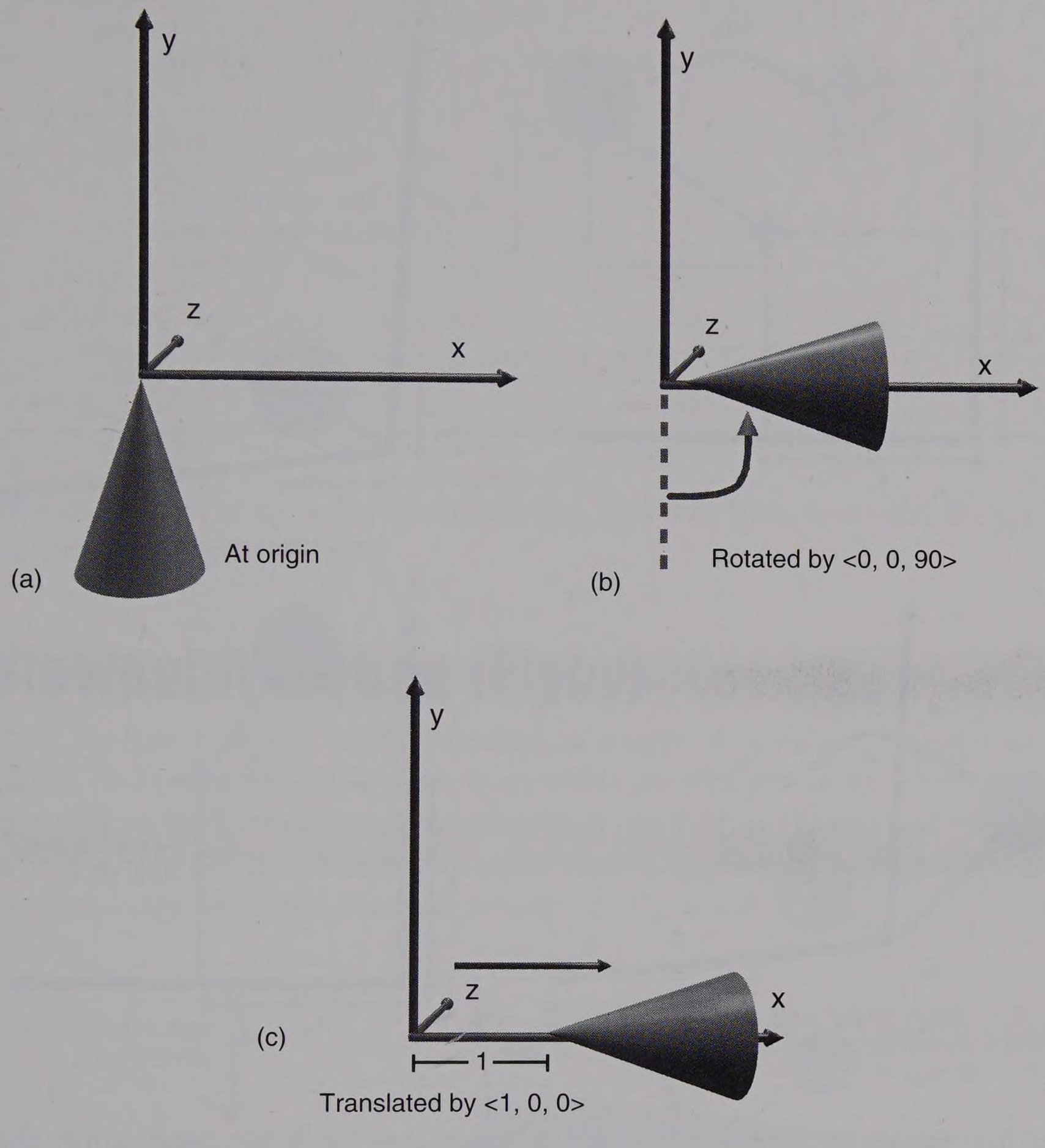


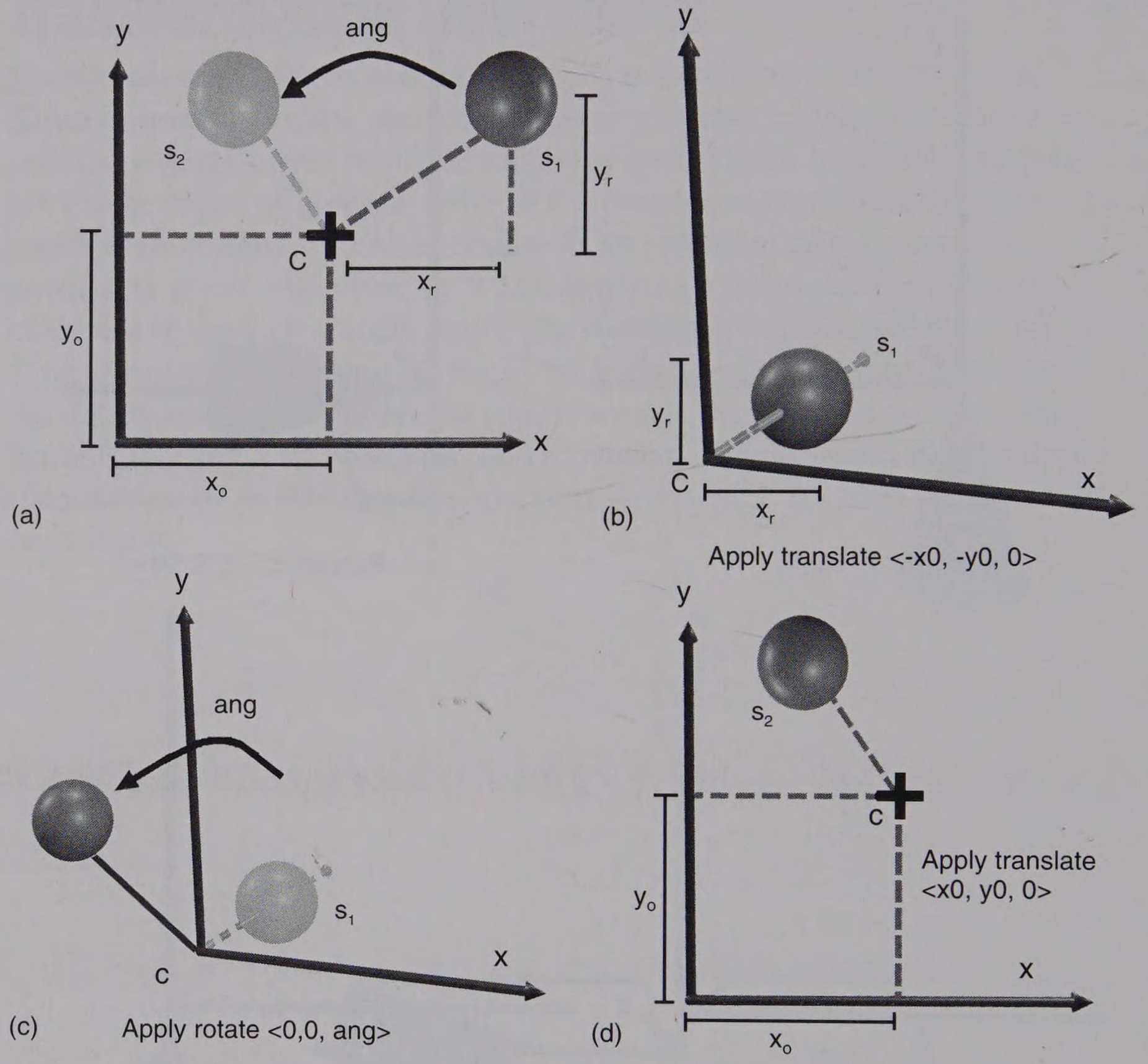
Figure 8.2
Applying a rotation and a translation to an object.

The main point to remember is that rotations are applied about a pivot point located at the origin. In animation, it is useful to create the object at the origin, rotate it, then apply the translations to locate the object in its final position.

Sometimes it is necessary to rotate an object using a pivot point located outside the origin (see fig 8.3a). In this example we want to rotate sphere S1 about the pivot point C by an angle "ang" about the Z axis. The pivot point C is located at <xo, yo, 0> and the sphere is located at <xo + xr, yo + yr, 0>.

Figure 8.3

Rotation of an object about a pivot point.



The first step is to move point C to the origin (see fig. 8.3b). Next, apply the rotation about the z axis (see fig. 8.3c). Finally, locate point C back to it's original place (see fig. 8.3d). As you can see, it's easy to bring the pivot point back to the origin, apply the rotations, then locate it back to its original place. In Polyray, the operations would be written as it is presented in Listing 8.1.

Listing 8.1. Figures 8.3a, b, c, and d in Polyray format (partial listing).

```
define xo ...
define yo ...
define xr ...
define yr ...
define ang ...
define radius ...
```

```
define S1
     object {
          sphere <xo + xr, yo + yr, 0>, radius
     }
define S2
     object {
          S1
          translate < -xo, -yo, 0>
          rotate <0, 0, ang>
          translate <xo, yo, 0>
     }
object {
     S2
     shiny_red
}
```

By leaving most of the numbers as variables, you only have to modify the top section of Listing 8.1.

Viewpoint Motion (Flyby)

Flyby motion is widely used in animating scenes of satellites passing near a planet and cockpit views as seen from inside an airplane or jet. This type of motion can be accomplished by modifying the vectors "from" and "at" from the viewpoint definition. In other words, the objects being looked at are stationary while the observer moves (see fig. 8.4).

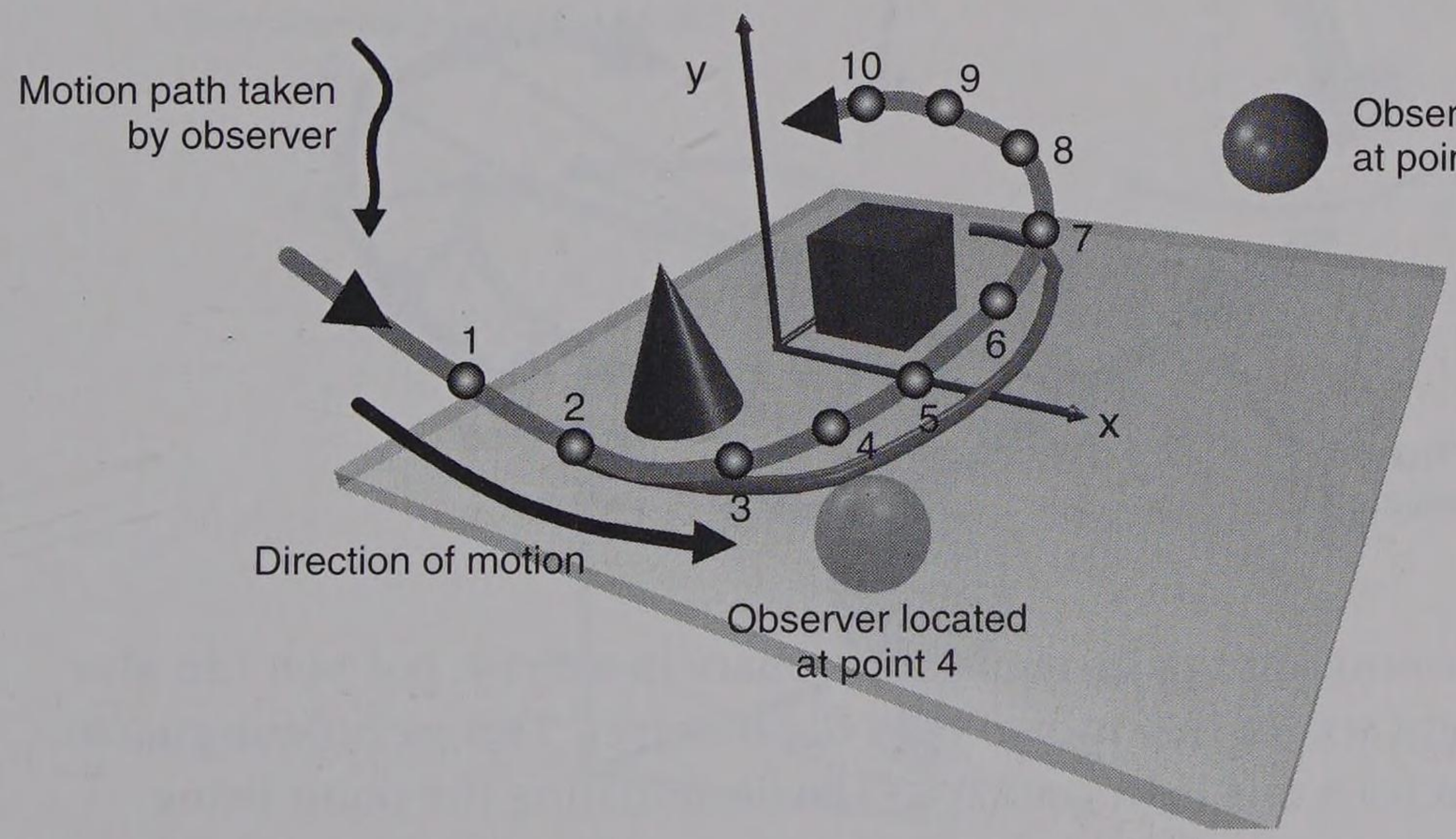


Figure 8.4
Path motion of an observer looking at the origin (flyby).

In figure 8.4, the observer follows a motion path. At each point, the point being looked at is the origin. A variation on this technique is to define the

motion of the observer plus the motion of the point being looked at (see fig. 8.5). In other words, there would be two motion paths: one for the observer's position and another for the "look at" point (see fig. 8.6). Figure 8.6 displays the motion paths without the objects.

Figure 8.5

Path motion of an observer looking at different points (flyby).

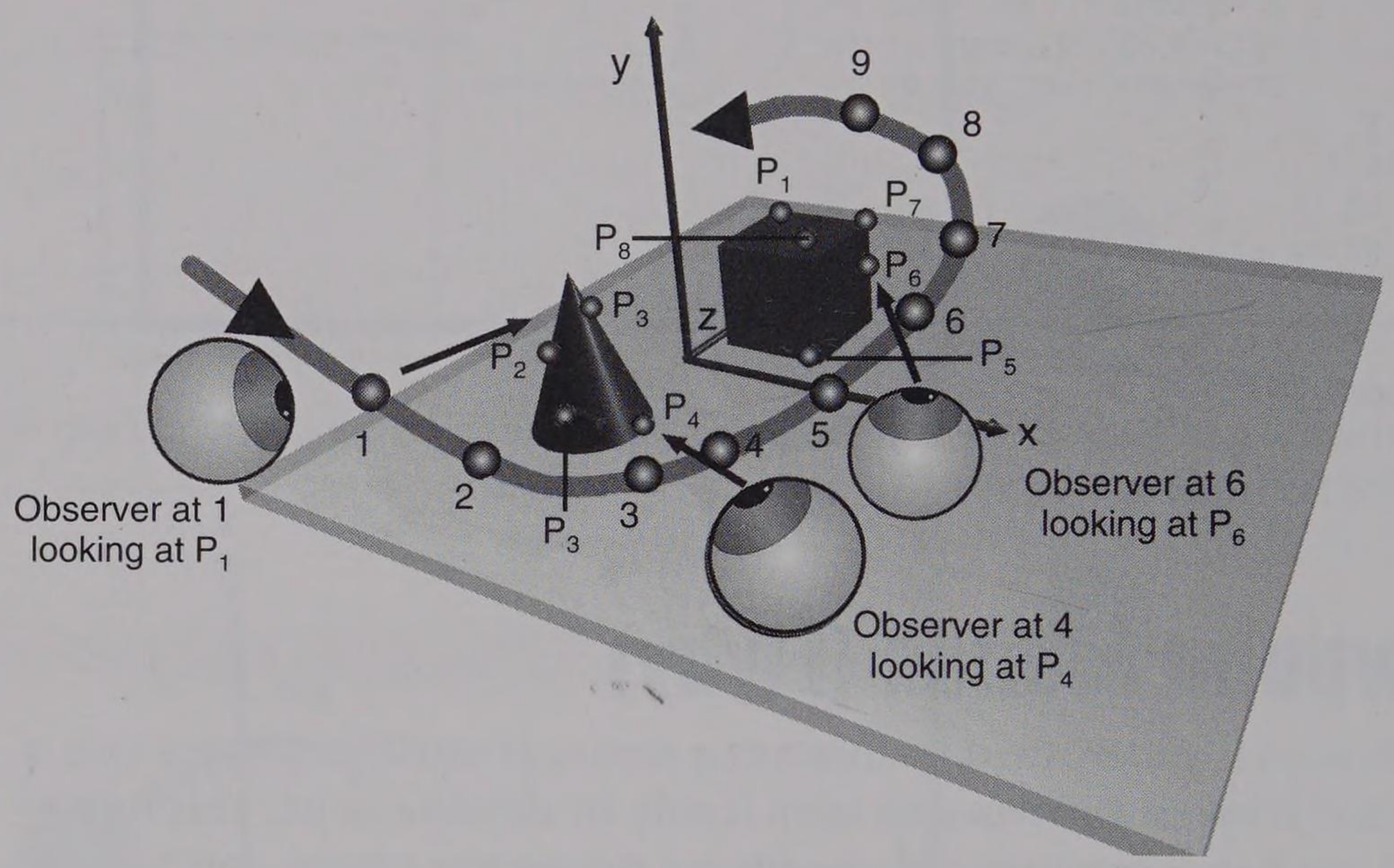
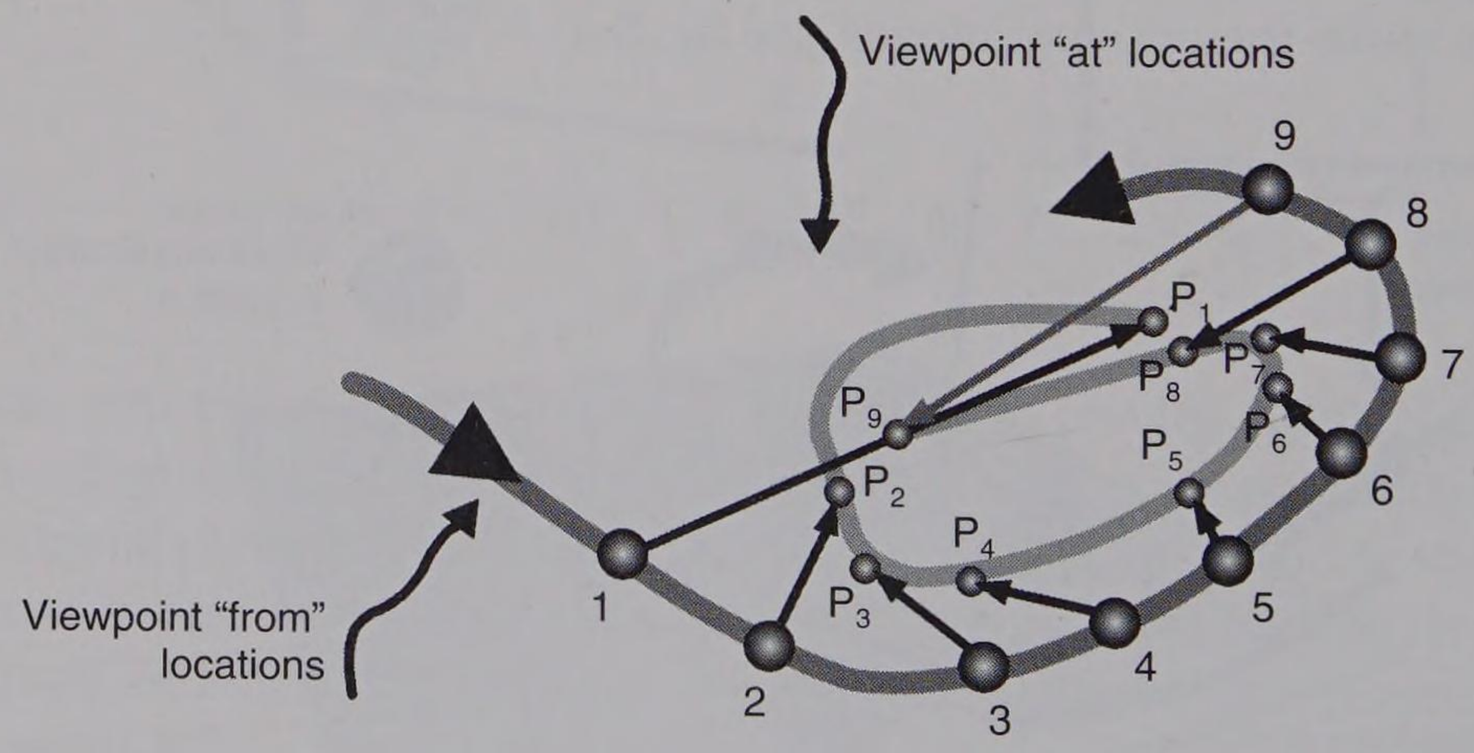


Figure 8.6

Path motion of the observer and the "look at" points.

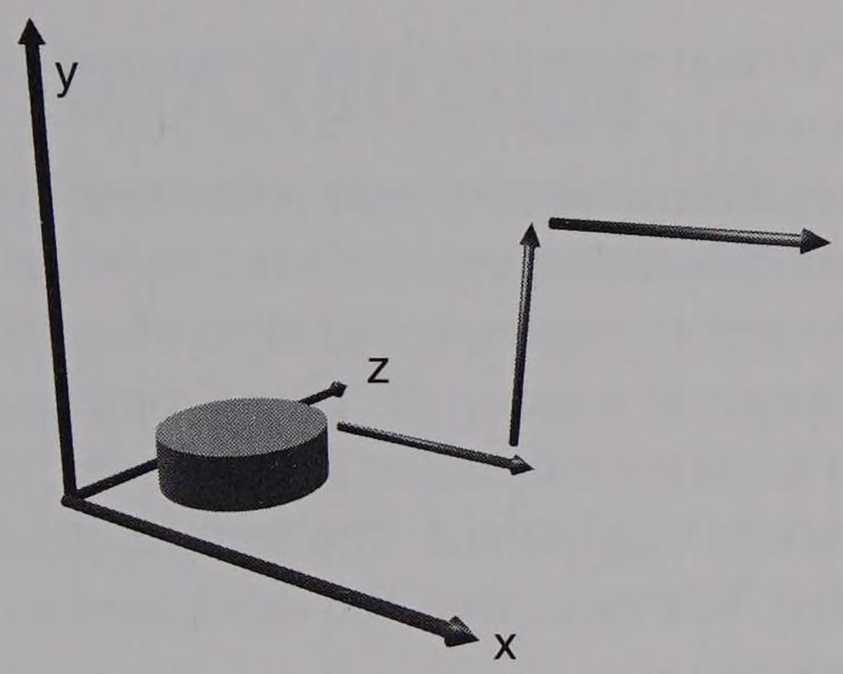


The light sources normally remain stationary in a flyby, but you can also create a light source that moves with the observer. This technique guarantees that at least one light source will be illuminating the point being looked at. If we assume that the "from" vector of the viewpoint definition is "from <x, y, z>," the light source can follow the observer by using the definition "light <x + dx, y + dy, z + dz>" where x, y, and z are the current coordinate locations of the observer and dx, dy, and dz are constant offset values.

As an analogy, the light source following the observer can be compared to the lamp on a miner's hat, which always follows the motion of the head.

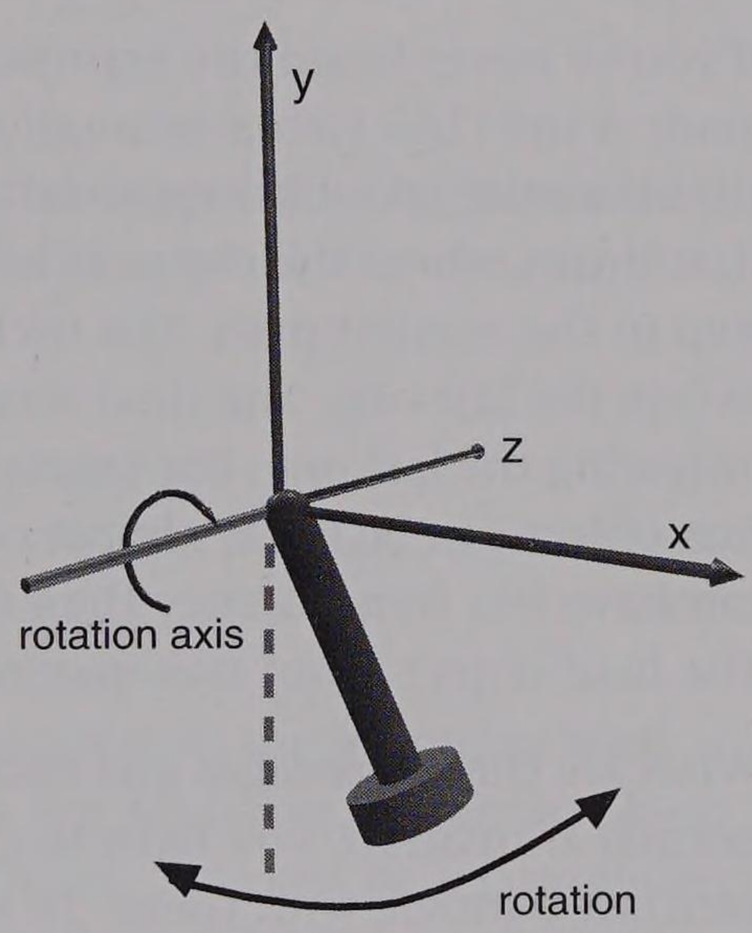
Object Motion

The motion of objects can be modeled by applying transformations directly on the objects. An object can have simple translation (hockey puck—see fig. 8.7a), rotation motion in x, y, and z or a combination of both. A spinning top is a good example of an object that combines rotation and translation (see fig. 8.7c). Figure 8.7 also presents two kinds of rotations: about an axis away from the object (pendulum—see fig 8.7b) and about an axis that passes through the object (spinning top—see fig. 8.7c).



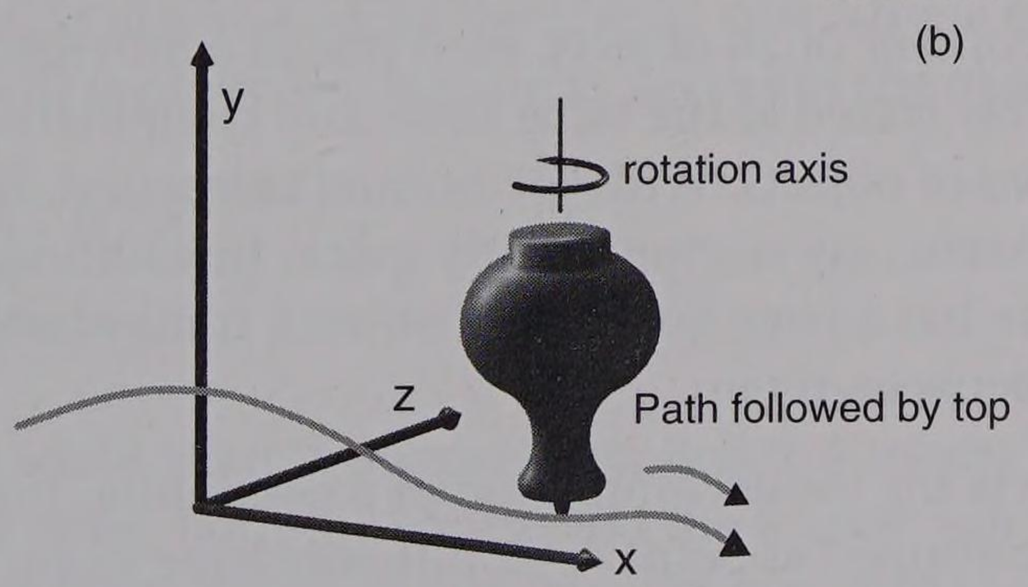
Pure translation "hockey puck"

(a)



"Pendulum"

(b)



Rotation & translation
"spinning top"

(c)

Figure 8.7

Examples of simple translation, rotation, and combined motions in objects.

It might be interesting to know that objects can appear to move if the viewpoint is not stationary. This technique requires that the viewpoint move in a way that makes the object look as if it is in motion. For example, to a person driving a car all the people on the street appear to move, even the people who are just standing on the sidewalk. The driver is a moving "viewpoint."

Finally, both the flyby and the object motion can be combined into one scene. For example, you can simulate a person walking as another person rides a bicycle and passes near him. Parts of the bicycle (gears, pedals) move, as does the observer.

Virtual Animation

If you've never heard the expression "virtual animation," it's because I just made it up! I use virtual animation to refer to a stationary scene that simulates the "double exposure" effect of cameras. You can create one image that shows where the object is in terms of location and orientation at each step in the motion path. The trick is to add transparency to all of the objects except the last one. The final scene appears as a group of "ghost" objects following the last one. For example, if a sphere animation has 20 steps, the first object can be made almost completely transparent. The second object can have less transparency than the first, but more that the third, and so on. The final object is not transparent (opaque).

What are the advantages and disadvantages of such a technique? With normal animation, you have to create 20 files (as an example) and then generate a movie with them. To see if the animation came out as expected, you must watch the movie. In virtual animation, only one file is ray traced, but this scene consists of 20 objects, each one in a different position. All of the objects are ray traced at the same time, and computation time increases with the number of objects. With the normal animation, if you only have one object per scene, ray tracing is fairly quick. In addition, if your virtual animation scene has a large quantity of objects, it may become too difficult to distinguish between them.

Figure 8.8 depicts the virtual animation of a pendulum. Initially, the pendulum is positioned at point A. At different time steps, it is shown as it swings towards point B. By looking at the variation of the transparency, we can determine the initial and final positions as well as the direction of the motion.

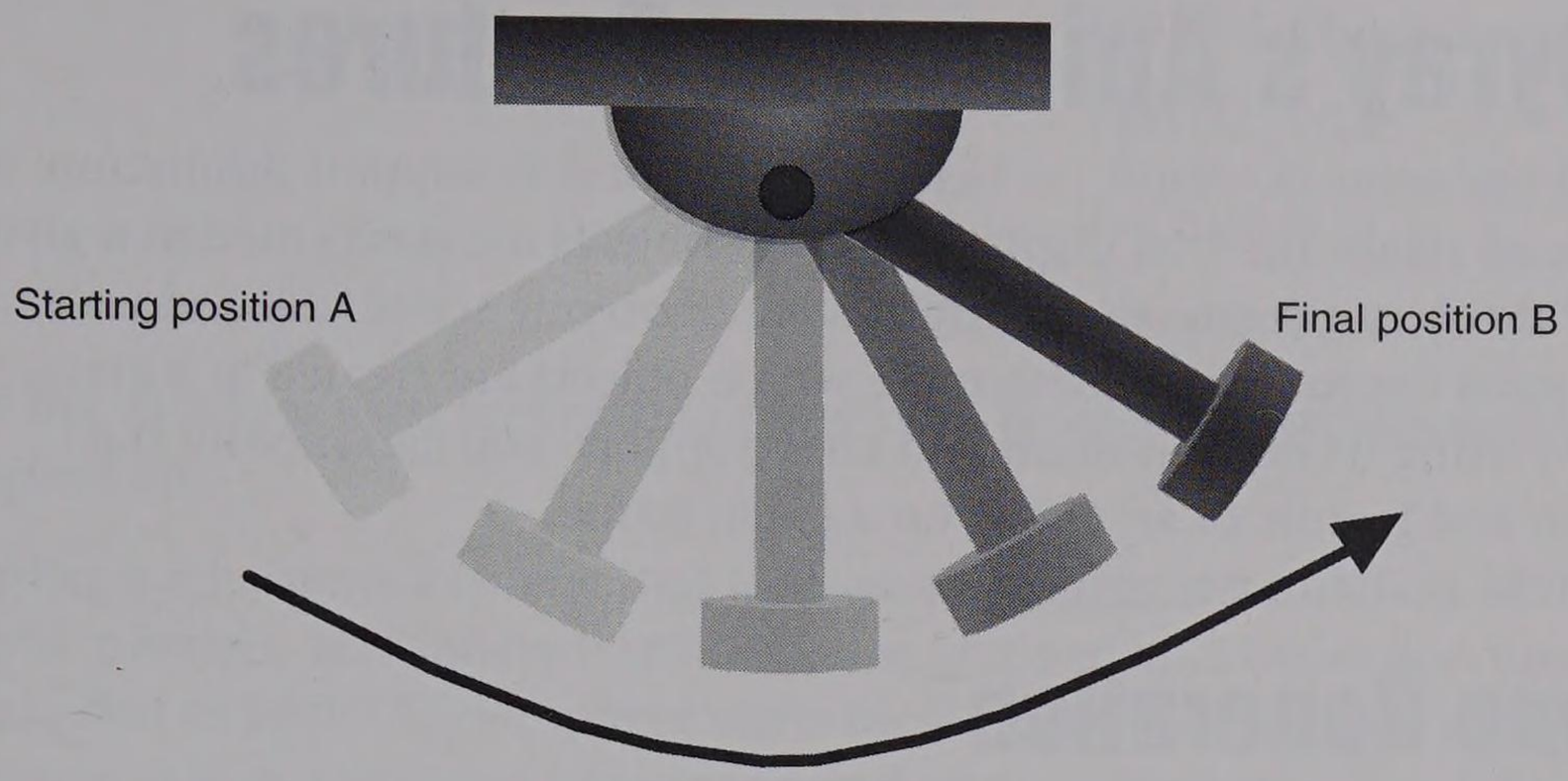


Figure 8.8
Virtual animation of a pendulum.

Texture Variation

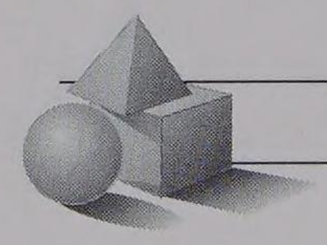
If you review the example scene files presented in previous chapters, you might note that the texture declaration is always applied after the transformations. For example,

```
object {
     sphere .....
     rotate ...
     scale ...
     texture...
}
```

This implies that the rotations are only performed on the geometry itself and not on the texture. In animation, this is not advisable. When an object moves, the texture should move correspondingly. This can be accomplished by applying the texture *before* applying the transformations you use to animate the object. When you apply the texture first, all the transformations that follow will apply to the texture as well as to the object:

```
object {
     sphere <0, 0, 0>, 2
     white_marble
     rotate <0, 45, 0>
}
```

In this example, a white_marble texture is applied to a sphere located at the origin. After this is done, both the sphere and the texture will be rotated 45 degrees about y. Since the sphere is symmetrical, you only see the texture rotate.



Polyray's Animation Features

Polyray has some powerful yet easy-to-use features to support animation. A *frame* is an image file that displays how the objects are positioned in a given time. Polyray can create a set of frames based on one scene file, and use conditional expressions to determine which objects are created at a given time. By using its built-in math and array support, you can modify the position and texture of an object on a given frame.

Frame Generation

If the position of the objects and the viewpoint is constant at each frame, the scene is referred to as "stationary." In animation, you want to do the opposite: change the position and orientation of the objects, the viewpoint, or both on each frame.

To create a set of frames with Polyray, you must indicate the number of frames to be generated, the starting number for the first frame, and either the ending number for the final frame or the total number of frames to be created. The syntax for frame creation is

`start_frame`	*value*	(variable) number of the first frame to be created
`end_frame`	*value*	(variable) number of the last frame to be created
`total_frames`	*value*	(variable) total number of frames to be created
`frame`		(variable) holds the current frame being ray traced
`outfile "`*filename*`"` or `outfile` *filename*		filename is the name of the frames to be created. The name must be five letters or less since Polyray appends three digits to the end of this name. **Example:** If the filename is TEST and you need 3 frames, starting at 1, Polyray will create:

TEST001.TGA

TEST002.TGA

TEST003.TGA

The default name is OUT.

All the variables used for creating frames can be used in mathematical expressions.

Listing 8.2 presents a sphere that increases its radius as a function of the frame number. The sphere is inside a unit box and an intersection operation is applied to both. As the sphere starts increasing in size beyond the box, the box acts as a clipping object; any part of the sphere that goes beyond the box is discarded. Figure 8.9 presents the four frames generated by ANIMCLIP.PI.

Listing 8.2. A sphere that increases in size(ANIMCLIP.PI).

```
//ANIMCLIP.PI
start_frame 1
end_frame 4
include "colors.inc"
include "texture.inc"
viewpoint {
 from      <0.0, 3.0, -6.0 >
 at        <0.0, 0.0, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
  light < 10 ,  10 , -20  >
// LIGHT_SOURCE
  light <-10 ,  -1 , -20  >
// LIGHT_SOURCE
  light <1 ,  10 , 0  >
background white
object {
    object {
        box <-1, -1, -1>, <1, 1, 1>
    }
*   object {
        sphere <0, 0, 0>, 0.5*frame
    }
    rotate <-17, 15, 0>
    wooden
}
```

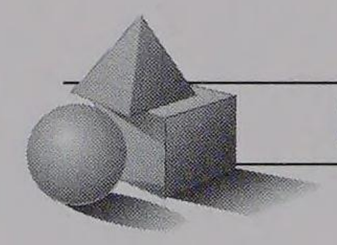
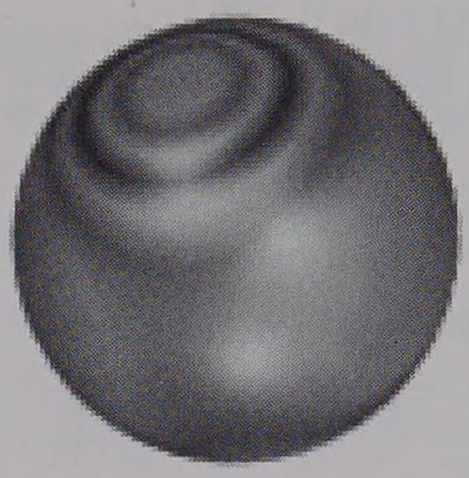
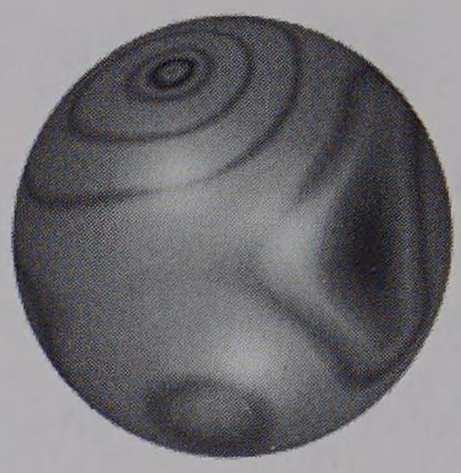


Figure 8.9

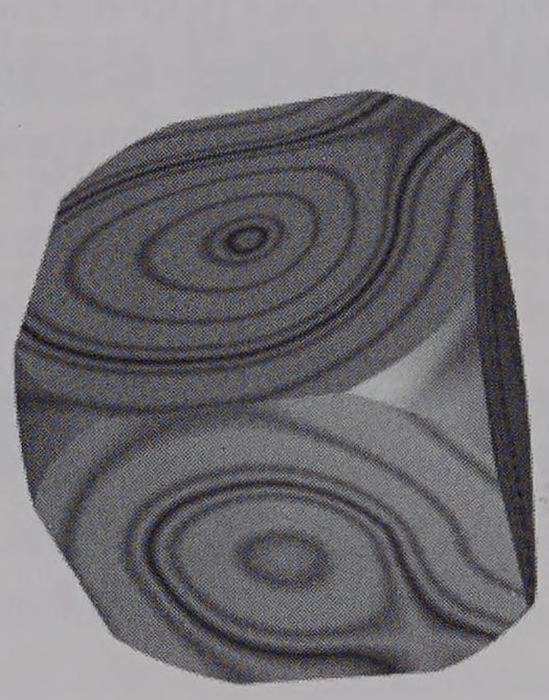
Sphere Animation (ANIMCLIP.PI).



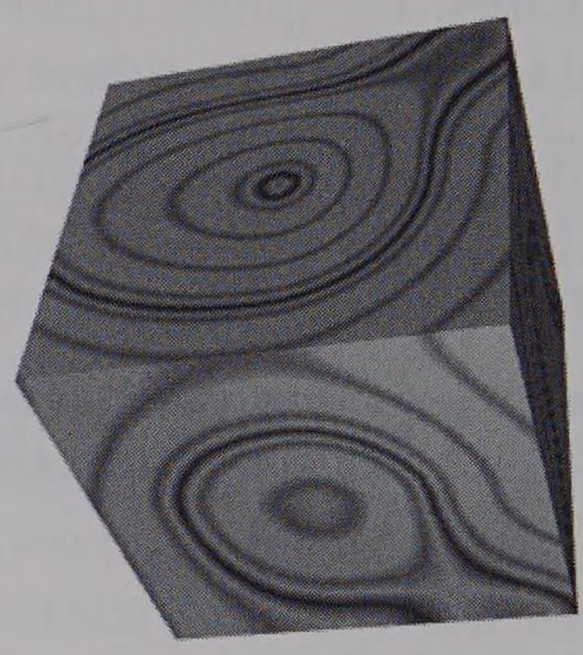
Frame 1



Frame 2



Frame 3



Frame 4

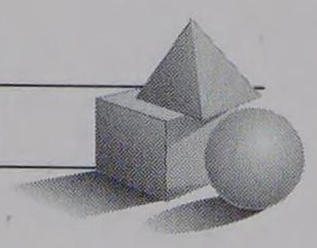
In Frames 1 and 2, the sphere is still smaller than the box. The radius of the sphere is

```
frame 1: 0.5 * frame = 0.5 * 1 = 0.5
frame 2: 0.5 * frame = 0.5 * 2 = 1.0
```

In Frame 3, parts of the sphere are clipped away and the shape appears more like a box. The radius in this frame is `0.5 * frame = 0.5 * 3 = 1.5`. Finally, Frame 4 presents a box with the texture used in the sphere. The sphere's radius is `0.5 * frame = 0.5 * 4 = 2`.

Notice that in Listing 8.2 the variable `total_frames` and the output file declaration were not used. Therefore, `total_frames = end_frame + 1`, and the filename defaults to OUT.

Conditional Processing



Conditional processing gives you the ability to define objects or token names based on the status of a conditional expression. For example,

```
if (frame == 0)
     define Radius 0.5
else
     define Radius 1.0
```

In this example, the token name `Radius` has a value depending on the current frame being ray traced. If `frame` is equal to zero, the `Radius` will be equal to 0.5, otherwise it will be equal to 1.0. You can also create objects in the same manner:

```
if (Radius == 1.0)
     object {
          sphere <0, 0, 0>, Radius
          shiny_red
     }
```

In this example, a shiny red sphere will be created on the current frame if `Radius` is equal to 1.0. If `Radius` is different from 1.0, the sphere will not appear on the scene. The syntax for conditional processing is

```
if (cexper) {
     [definition #1a]
     [definition #2a]
     ...etc...
}
else {
     [definition #1b]
     [definition #2b]
     ...etc...
}
```

where *cexper* is a conditional expression (refer to Chapter 6, Conditional Expression Section)

definition can be of the type:

object {...}

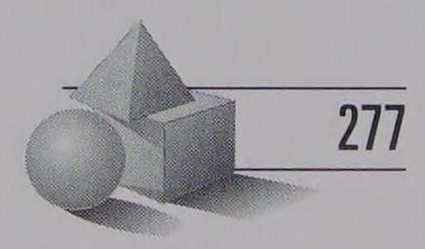
light ...

define ...

The curly brackets are used only if there is more than one definition inside the `if` or the `else` statements.

You can also nest conditional processing expressions:

```
if (cexper #1) {
       ...etc...
}
else  if (cexper #2) {
       ...etc...
      }
```



```
else {
...etc...
}
```

As a final note, you can have complete control over objects in animation by combining conditional processing, conditional expressions, and special surface textures.

Static Definitions

As you know, conditional processing can create objects depending on a conditional expression:

```
if (frame == 0)
     object {
          sphere <0, 0, 0>, 2
          shiny_red
     }
```

The only problem with this expression is that the sphere will only be seen when `frame = 0`. On the remaining frames, the sphere will not appear. In other words, if you wish the object to appear on all of the frames you must replace the conditional expression with `frame >= 0`:

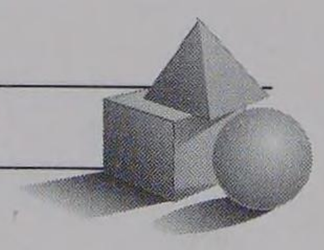
```
if (frame >= 0)
     object {
          sphere <0, 0, 0>, 2
          shiny_red
     }
```

or simply discard the conditional processing expression

```
     object {
          sphere <0, 0, 0>, 2
          shiny_red
     }
```

The `static` keyword is used to declare that once an object is defined, it does not have to be redefined on all the remaining frames; it will remain static. The declaration

```
if (frame = 0)
        static define MySphere
         object {
                sphere <0, 0, 0>, 2
                white_marble
         }
 object {
         MySphere
         rotate <0, 10*frame, 0>
 }
```



indicates to Polyray that the sphere called `MySphere` will be created once at the beginning (frame = 0) and will stay in memory throughout the creation of all the remaining frames. Using this on a sphere doesn't make much sense, but if you have a large quantity of objects, or those objects require lots of computation time (height fields, for example), Polyray will do the computation only once and save the objects to memory. In other words, all the tough computations are done once and then the remaining frames will just use the objects available in memory.

A `static define` can also be used to save and modify the values of variables from frame to frame:

```
if (frame == start_frame)
        static define radius 1.0
else    if (frame == start_frame + 1)
                static define radius (radius - 0.5)
        else
                static define radius (radius - 0.3)
```

In this example, radius is given the value of 1.0. In the next frame, `radius = radius -0.5` and the value is saved in memory. For the remaining frames, `radius = radius -0.3`.

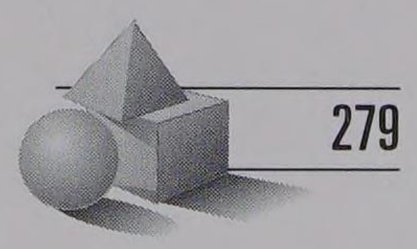
Arrays

Arrays are a powerful tool in animation. A typical example is the use of a path data file that contains the locations of the object on each frame. The frame number can be used as the index of the array to extract the current position:

```
start_frame 0
end_frame 3

define Location         [<0, 0, 0>,
                         <0.3, 5.5, 2.1>,
                         <0, -5, 3>,
                         <-6, 4, -1>]

object {
        sphere Location[frame], 2
        shiny_red
}
```

where, as an example, `Location[0] = <0, 0, 0>` and `Location[1] = <0.3, 5.5, 2.1>`.

Listing 8.3 presents a program that reads a data file and converts it into a Polyray array. The program is compatible with QBasic and Quick Basic 4.5.



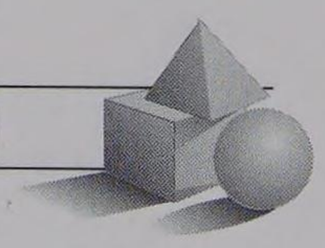
Listing 8.3. Program to read a data file and create a Polyray array.

```
' ARRAY.BAS
' PROGRAM TO READ A DATA FILE AND CREATE AN ARRAY FOR POLYRAY
'
' This Program reads a data file that consists of 3D points in the form
' p1.x p1.y p1.z
' p2.x p2.y p2.z
' ...etc...
DIM inf AS STRING, outf AS STRING, an AS STRING
DIM x AS SINGLE, y AS SINGLE, z AS SINGLE
CLS
INPUT "Enter Name of Data File [TEST.DAT]"; inf$
IF inf$ = "" THEN inf$ = "TEST.DAT"
PRINT
INPUT "Enter Name of Output File [ARRAY.OUT]"; outf$
IF outf$ = "" THEN outf$ = "ARRAY.OUT"
PRINT
INPUT "Enter Token Name for Array [MyArray]"; an$
IF an$ = "" THEN an$ = "MyArray"
OPEN inf$ FOR INPUT AS #1
OPEN outf$ FOR OUTPUT AS #2
PRINT #2, "// Array ["; an$; "] created with ARRAY.BAS"
PRINT #2, "// from data file ["; inf$; "]"
PRINT #2, ""
PRINT #2, "define "; an$; " ["
DO
 INPUT #1, x, y, z
 PRINT #2, "              <"; x; ", "; y; ", "; z; ">";
 IF NOT EOF(1) THEN
 PRINT #2, ","
 END IF
LOOP UNTIL (EOF(1))
PRINT #2, ""
PRINT #2, "         ]"
CLOSE #1
CLOSE #2
```

The program asks the name of the data file to be used as input, the name of the output file to be created, and the token name for the array. The file can the be added to your scene files with a text editor or using an `include` statement.

To test the program, create an ASCII file with your text editor and type in the following data as it appears (press Enter after each line). Save the file as TEST.DAT:

```
1 1 1
2 2 2
3 3 3
4 4 4
5 5 5
```



Run ARRAY.BAS and press enter to accept all the default values. Listing 8.4 presents the output generated by ARRAY.BAS.

Listing 8.4. Output generated by ARRAY.BAS (ARRAY.OUT).

```
// Array [MyArray] created with ARRAY.BAS
// from data file [TEST.DAT]

define MyArray [
        < 1 ,  1 ,  1 >,
        < 2 ,  2 ,  2 >,
        < 3 ,  3 ,  3 >,
        < 4 ,  4 ,  4 >,
        < 5 ,  5 ,  5 >
      ]
```

The output file ARRAY.OUT presents the name of the array [MyArray] and the data file used as input [TEST.DAT]. To refer to this file in your scene, use the statement `include "array.out"`.

Creating the Animation File

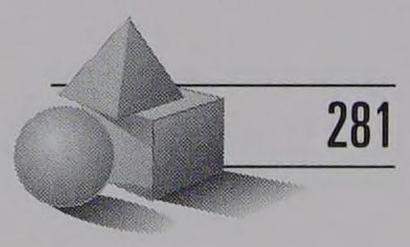
The animation file is created by an external utility called DTA (Dave's .TGA Animator) written by David K. Mason (check the Introduction of this book for bulletin boards where you can find DTA). DTA creates Autodesk Animator-compatible files with the .FLI extension. The program accepts a series of command line parameters that define the files to be used in order to create the animation file, the speed of the animation and the output file name. DTA does not play the animation file after it generates it. You'll need another program called AAPLAY (Autodesk Animator Player) to view it.

For our purpose, DTA can be executed simply as

DTA NAME*.TGA

where NAME is the 5 letter name given to the frames created by Polyray. Since Polyray will generate the frames in numerical sequence (NAME000.TGA, NAME001.TGA, NAME002.TGA, and so on) using the wildcard symbol (*) will tell DTA to select all the files that start with the word NAME and have the extension TGA. DTA will sort the files in the correct order. Since we didn't define a name for the output file, the default "ANIM.FLI" is used. If you wish to change this name, execute DTA as

DTA NAME*.TGA /OFILENAME.FLI



where FILENAME is the name for the output file. The animation files should always end with the FLI extension.

Once the animation file is created, run AAPLAY and select your animation file from the menu to execute it.

Resolution Issues

There are two important issues concerning the resolution of the images to be ray traced and animated. First, creating multiple frames may consume plenty of time and hard drive space. As a suggestion, use low resolution in your viewpoint definition, for example, resolution 100,100 with an aspect ratio of 1.0. This reduces computation time and space requirements. Generate the animation file and preview it for correctness. If all is well, change to a higher resolution. Second, the highest resolution for a FLI animation is 320 by 200. If you wish to generate higher resolutions, use DTA to create FLC files (but you'll need an FLC player to view the animation, plus a good graphics card and a fast machine!).

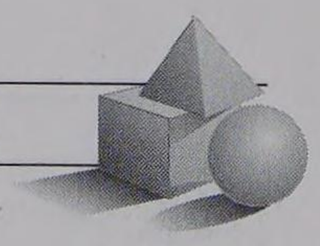
When generating the final animation, use anti-aliasing, since aliasing is more noticeable at low resolutions (320 × 200).

Animation Examples

In the following section you'll have the opportunity to apply the different types of animation techniques presented in the previous sections: flyby, object, virtual, and texture variation.

Flyby

One of the easiest examples of flyby animation is to follow the path of a satellite as it orbits a planet. In order to compare the cases of the viewpoint when it's stationary and when it's in motion, you'll develop an animation that consists of two parts. The first part will have a planet with its own moon orbiting around it. The observer will remain stationary. In the second part, both the planet's moon and the observer are in motion.



To understand the motion of the moon and the observer, refer to figure 8.10.

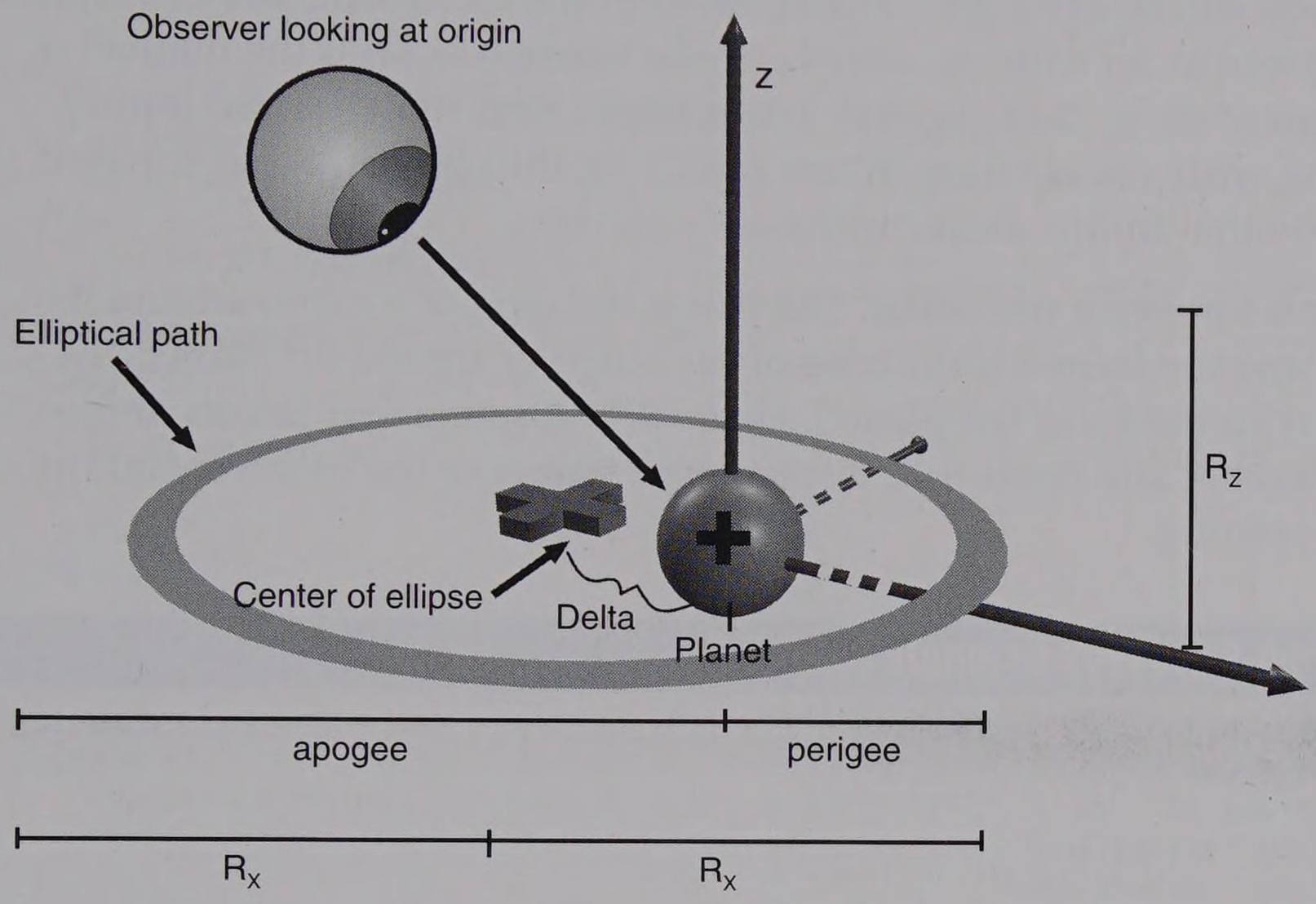


Figure 8.10
Elliptical orbit.

Figure 8.10 presents an elliptical path for the orbit. The smallest distance between the observer and the planet is called the *perigee,* and the largest distance is called the *apogee*. The *delta* is the difference between the center of the elliptical orbit and the center of the planet. The ellipse can be described by

$$x = Rx * \cos(2pt)$$
$$z = Rz * \sin(2pt)$$

where $p = 3.1415....$

t varies from 0 to 1

$Rx = (\text{perigee} + \text{apogee})/2$

When Rx = Rz, the functions describe a circular path. (A good exercise is to use grid paper and calculate at least 4 points. Use t = 0, t =0.25, t = 0.5, t = 0.75, and connect them with lines.) In order to locate the center of the ellipse to the left of the planet (as it is in figure 8.10), the x value must be modified to

```
x = Rx * cos(2pt) - delta
```

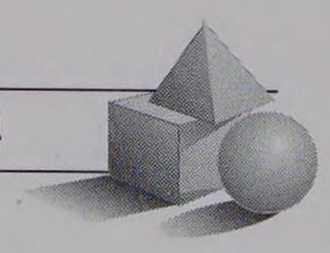
where `delta = ¦Rx - perigee¦`

Since you will be using the variable frame as the increment, you can replace the expression `2pt` with `(2p/sections)*frame` where `sections` is the number of divisions of the `2p` (360 degrees). For example, to divide the orbit into 15 sections, write `(2p/15)*frame`. When `frame = 15`, the observer has completed one full orbit. In this scene, define `div = (2p/15)`.

Listing 8.5 presents two orbits. The first is the orbit of a moon around its planet and the second is the orbit of the observer around the planet (the moon is also orbiting the planet). Figure 8.11 displays four frames of the moon's orbit, and figure 8.12 displays four frames of the observer and the moon orbiting.

Listing 8.5. Flyby example (ORBIT.PI).

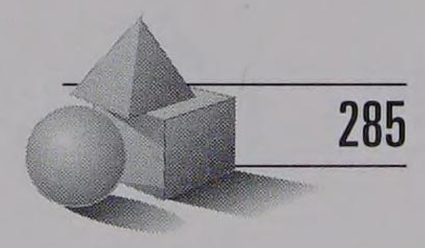
```
//ORBIT.PI
start_frame 1
end_frame 45
include "colors.inc"
include "texture.inc"
// Do not Modify
static define Pi 4*atan(1.0)      // Pi = 3.1415....
static define TwoPi 2*Pi          // 2*Pi
// Modify the following (if necessary)
static define perigee  1
static define apogee   3
static define Rz         2

// Do not modify
static define Rx          (perigee + apogee)/2
static define delta       fabs(Rx - perigee)
static define div         (TwoPi/15)
static define FirstCycle      <Rx - delta, 0, Rz>            // Stationary
static define SecondCycle     <Rx*cos(-div*frame) - delta,   // Orbit Motion
                               0,
                               Rz*sin(-div*frame)>

if (frame<= 15) {
        viewpoint {
        from      FirstCycle
        at        <0.0, 0.0, 0.0 >
        up        <0.0, 1.0, 0.0 >
        angle      45
        resolution 100,100
        aspect     1.0
        }
}
else {
viewpoint {
        from      SecondCycle
```



```
        at          <0.0, 0.0, 0.0 >
        up          <0.0, 1.0, 0.0 >
        angle        45
        resolution 100,100
        aspect      1.0
        }
}
// LIGHT SOURCE THAT FOLLOWS THE OBSERVER
if (frame<= 15)
        light FirstCycle
else
        light SecondCycle

// LIGHT_SOURCE
  light <-10 ,  -1 , -20  >
// LIGHT_SOURCE
   light <0, 20, 0>

// PLANET
object {
    sphere <0, 0, 0>, 0.3
    white_marble
}
// MOON
object {
    sphere <0.5*cos(div*frame),0, 0.5*sin(div*frame)>, 0.05
    shiny_yellow
}
```

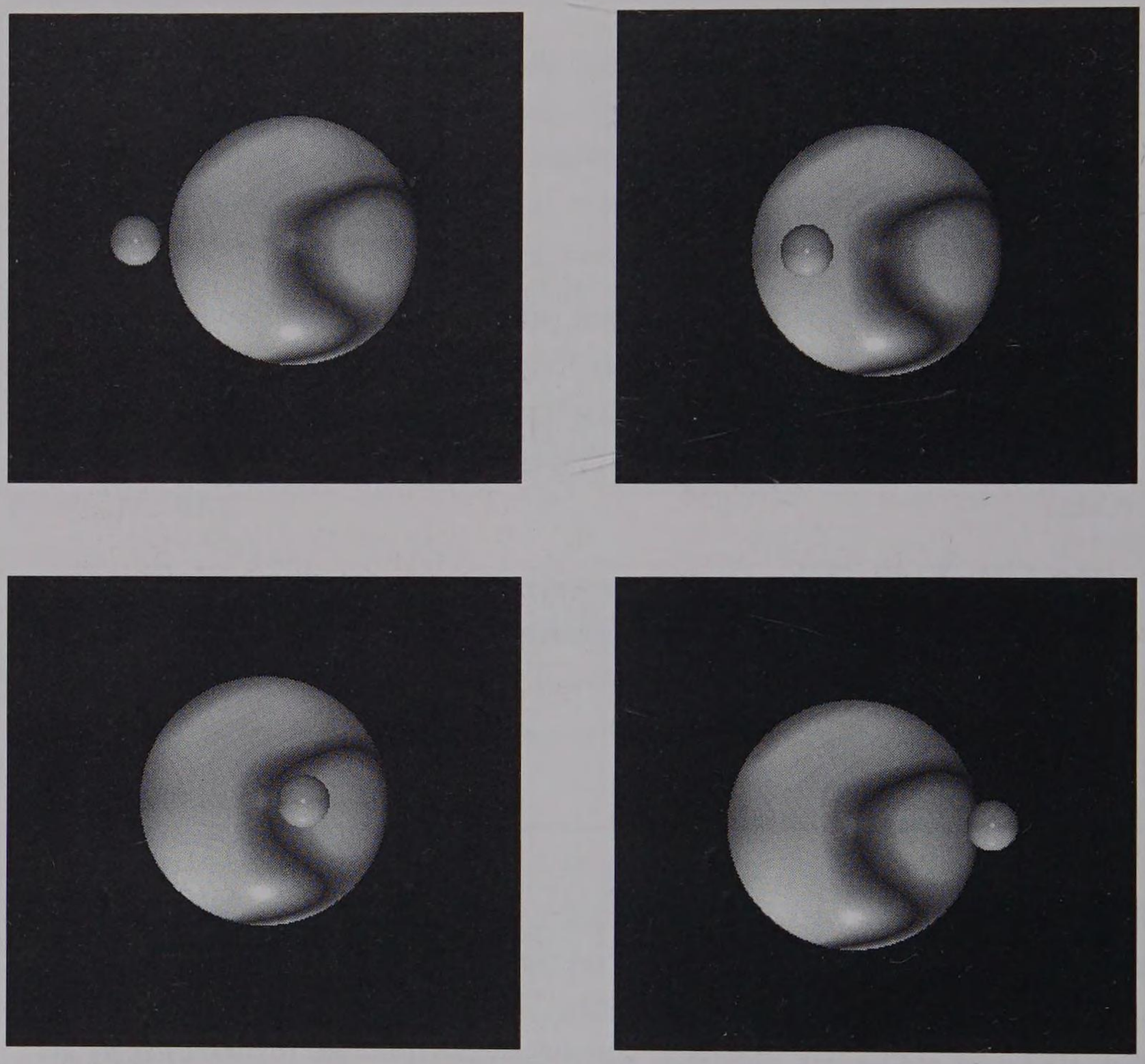
ORBIT.PI uses some interesting techniques. First, it has two viewpoint definitions depending on the number of the frame. This is accomplished through the use of conditional processing. If the frame number is less than or equal to 15, it keeps the observer stationary; otherwise it moves it in an elliptical path:

```
if (frame <= 15) {
        viewpoint {
        from      FirstCycle
        at          <0.0, 0.0, 0.0 >
        up          <0.0, 1.0, 0.0 >
        angle        45
        resolution 100,100
        aspect      1.0
        }
}
else {
viewpoint {
        from      SecondCycle
        at          <0.0, 0.0, 0.0 >
        up          <0.0, 1.0, 0.0 >
        angle        45
        resolution 100,100
        aspect      1.0
        }
}
```



The at vector remains constant in both cases.

Figure 8.11
Orbit animation (ORBIT.PI).



Another feature is the motion of one of the light sources. It follows the observer:

```
if (frame<= 15)
        light FirstCycle
else
        light SecondCycle
```

As I mentioned before, attaching a light source to the observer guarantees that at least one light source is illuminating what the observer is looking at.

Finally, the moon follows a circular orbit around the planet (Rx = Rz = 0.5), and it is continually in motion throughout the animation

```
object {
    sphere <0.5*cos(div*frame),0, 0.5*sin(div*frame)>, 0.05
    shiny_yellow
}
```

Can you distinguish a difference between figure 8.11 and figure 8.12? In the first four, the moon is the only object in motion. Since the planet is stationary, the texture doesn't rotate. In the four frames of figure 8.12, the observer is in motion, so you are able to look at different parts of the planet and the texture appears to rotate—not because the planet is in motion, but because the observer is.

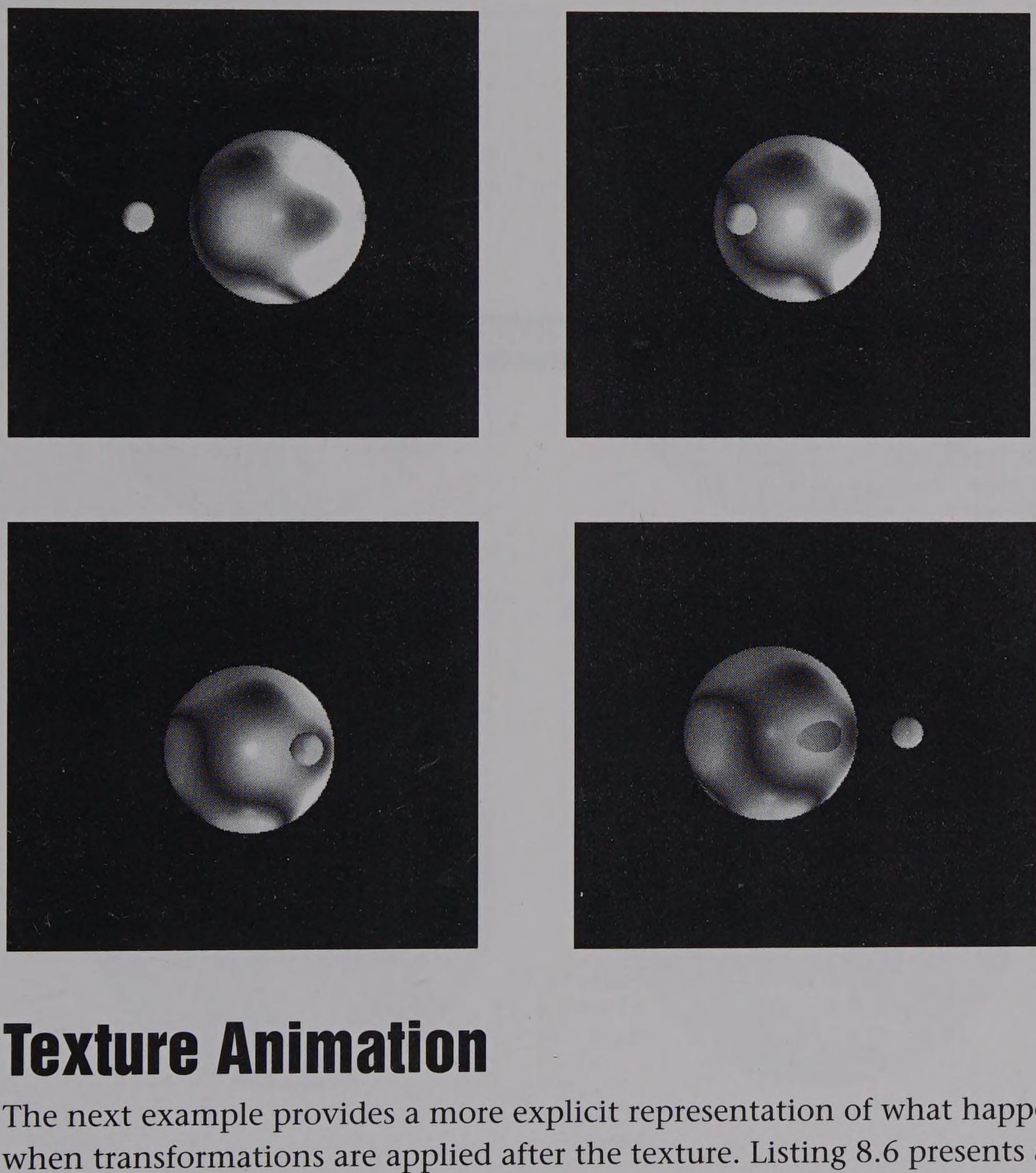


Figure 8.12
Orbit animation (ORBIT.PI).

Texture Animation

The next example provides a more explicit representation of what happens when transformations are applied after the texture. Listing 8.6 presents a sphere mapped with an alpha_yellow_marble texture. This texture was used in Chapter 4, "Layered Texture" section, and consists of a marble-like finish with some areas being transparent.

To animate the texture, the sphere and textures are rotated from 0 to 360 degrees about the y axis in 10 degree increments:

```
object {
   sphere <0,0,0>, 1.3
   alpha_yellow_marble
   rotate <0, 10*frame, 0>
}
```

Figure 8.13 displays the first four frames of AXISORBT.PI. To find out what happens when a transformation is placed before the texture, I suggest you modify Listing 8.6 by moving the statement rotate `<0, 10*frame, 0>` to before alpha_yellow_marble

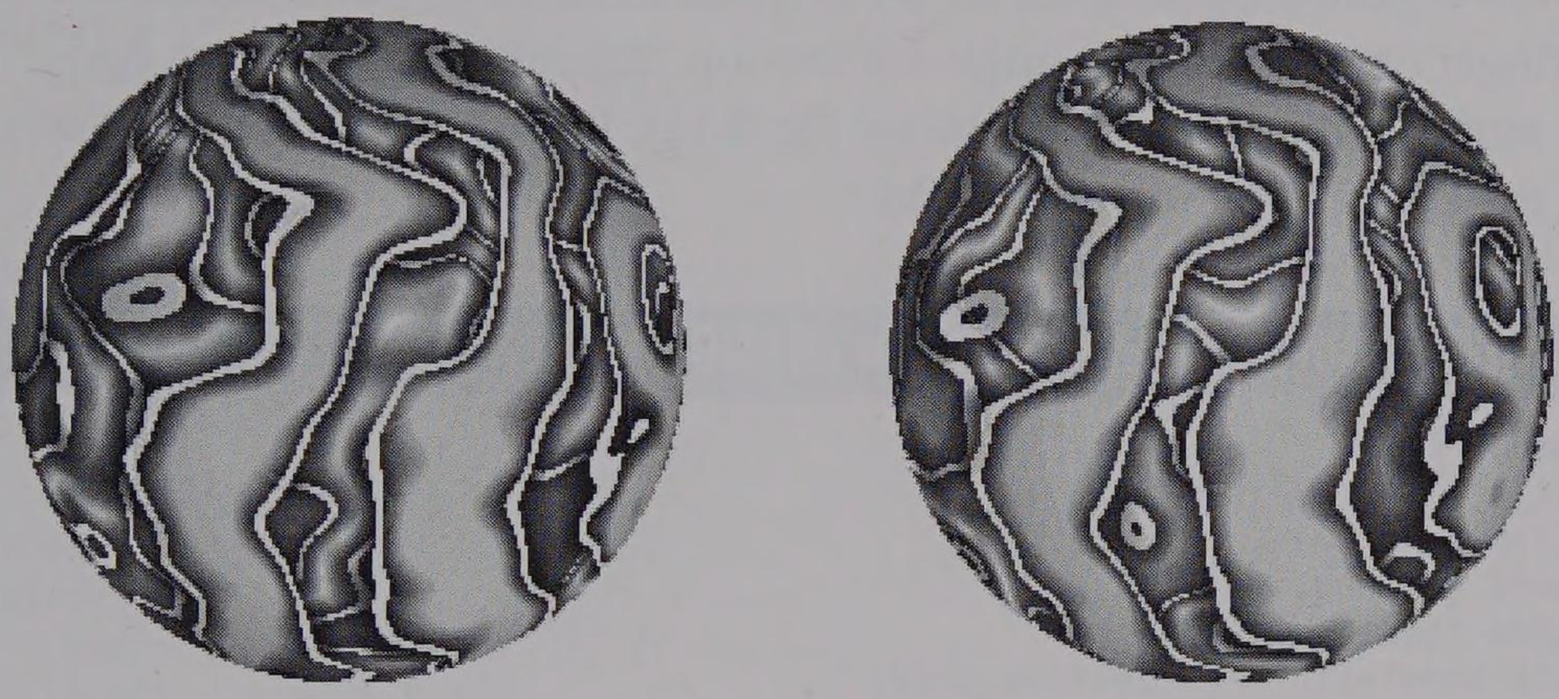
```
object {
   sphere <0,0,0>, 1.3
   rotate <0, 10*frame, 0>
   alpha_yellow_marble
}
```

and generate the animation again.

Listing 8.6. Rotating the texture (AXISORBT.PI).

```
// AXISORBT.PI
start_frame 0
end_frame  36
include "texture.inc"
include "colors.inc"
background white
define alpha_yellow_marble
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn   lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      transmission white, 1,1
      color_map(
       [0.0, 0.2, yellow, 0,   yellow, 0]
       [0.2, 0.5, yellow, 0,   black, 0.2]
       [0.6, 1.0, black, 0.2, black, 1]
      )
      }
}
viewpoint {
 from      <-2, 3, -7 >
 at        <0.0, -0.2, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
```

```
 resolution 100,100
 aspect     1.0
}
// LIGHT_SOURCE
directional_light < 0 ,  5 , -7  >
object {
   sphere <0,0,0>, 1.3
   alpha_yellow_marble
   rotate <0, 10*frame, 0>
}
```



Frame 1

Frame 2

Frame 3

Frame 4

Figure 8.13

Texture animation (AXISORBT.PI).

Object Animation

The following example models a robot hand and presents an animation of it catching a ball. This is a good example of how the position and orientation of objects can be dependent on the position and orientation of other objects. Take your own right hand for example. The index finger is divided into three parts: the base of the finger, the middle part, and the tip. If you can bend the tip, the other two sections of the finger stay put. If you bend the middle part, the tip will follow. And if you move the base of the finger, the middle part and the tip will follow (figure 8.14 presents all the steps described). In other words, the base of the finger depends on one angle for motion, the middle section on two angles, and the tip on three angles (see fig. 8.15). Listing 8.7 presents the scene file for the animation of a robot hand.

Listing 8.7. Robot hand animation (HAND.PI).

```
// HAND.PI
start_frame 0
end_frame 45
include "colors.inc"
include "texture.inc"
define HandTexture reflective_grey
define BallPath [                                   // MOTION OF THE BALL
                < 0 ,  0 , -19.2 >,
                < 0 ,  3.2 , -14.4 >,
                < 0 ,  3.2 , -11.2 >,
                < 0 ,  3.2 , -9.6 >,
                < 0 ,  1.6 , -8 >,
                < 0 ,  0.1 , -7.1 >,
                < 0 , -1.3 , -6.4 >,
                < 0 , -2.9 , -5.6 >,
                < 0 , -5 , -4.7 >,
                < 0 , -4 , -4.1 >,
                < 0 , -2.8 , -3.4 >,
                < 0 , -1.7 , -2.8 >,
                < 0 , -0.6 , -2.3 >,
                < 0 ,  0.2 , -1.9 >,
                < 0 ,  1 , -1.5 >
          ]

define ThumbAngley    frame*2                       // USE FRAME AS THE ANGLE.
define ThumbAngle2x   -frame                        // START AT 0 AND END AT
define ThumbAnglez    -frame                        // 45 DEGREES, EXCEPT THUMB
define FingerAnglex123 -frame
define FingerAnglex23  -frame
define FingerAnglex3   -frame
define FingerAnglez    0

define Plane_XZ
object { polygon 4, <-20,0,20>, <-20,0,-20>, <20,0,-20>, <20,0,20> }
// LIGHT_SOURCE
```

```
light < 10 ,  10 , 0  >
// LIGHT_SOURCE
light < -1 ,  5 , -20  >
viewpoint {
 from      <2.0, 5.0, -10.0 >
 at        <0.0, 1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      45
 resolution 100,100
 aspect     1.0
}

// OBJECT TO GRASP  - BALL STARTS MOTION AT FRAME #31
if (frame>30)
object {
   sphere BallPath[frame - 31], 1
   shiny_red
}
// FLOOR
object {
     Plane_XZ
     translate <0, -6, 0>
     matte_brown
}
// DEFINE THE PALM
// BOX
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 1.35,1.5,0.5>
    HandTexture
}

//DEFINE THE THUMB
define ThumbPart
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 0.3,0.5,0.5>
}
object {
// BOX THUMB 1
    object {
        ThumbPart
        translate<0.0, 0.5, 0.0>
     }
+
// BOX THUMB 2
     object {
         ThumbPart
         translate<0.0, 0.5, 0.0>
         rotate<ThumbAngle2x, 0.0, 0.0>
         translate< 0.0, 1.0, 0.0>
   }
   rotate < 0.0, 0.0, ThumbAnglez>
   rotate < 0.0, ThumbAngley, 0.0>
   translate < 1.75,0.0,0.0>
   HandTexture
}
```

continues

Listing 8.7. Continued.

```
// DEFINE A GENERIC FINGER
define FingerPart
object {
    box <-1, -1, -1>, <1, 1, 1>
    scale < 0.3,0.5,0.5>
}
define WholeFinger
object {
// FINGER PART 1  (NEAREST TO THE PALM)
object {
    FingerPart
    translate<0.0, 0.5, 0.0>
}
+
object {
// FINGER PART 2  (MIDDLE SECTION)
object {
    FingerPart
    translate<0.0, 0.5, 0.0>
}
+
// FINGER PART 3  (TIP)
    object {
        FingerPart
        translate<0.0, 0.5, 0.0>
        rotate<FingerAnglex3, 0.0, 0.0>
        translate<0.0, 1.0, 0.0>
    }
    rotate<FingerAnglex23, 0.0, 0.0>
    translate < 0.0 , 1.0, 0.0>
}
    rotate<FingerAnglex123, 0.0, FingerAnglez>
}

// CREATE INDIVIDUAL FINGERS
// INDEX
object {
   WholeFinger
   scale <0.9, 0.8, 1>
   translate < 1.1, 1.75,0.0>
   HandTexture
}
// MIDDLE
object {
   WholeFinger
   scale <0.9, 1, 1>
   translate < 0.3, 1.75,0.0>
   HandTexture
}
// RING
object {
   WholeFinger
   scale <0.9, 0.9, 1>
   translate < -0.5, 1.75,0.0>
   HandTexture
```

```
}
// PINKY
object {
   WholeFinger
   scale <0.9, 0.75, 1>
   translate < -1.3, 1.75,0.0>
   HandTexture
}
```

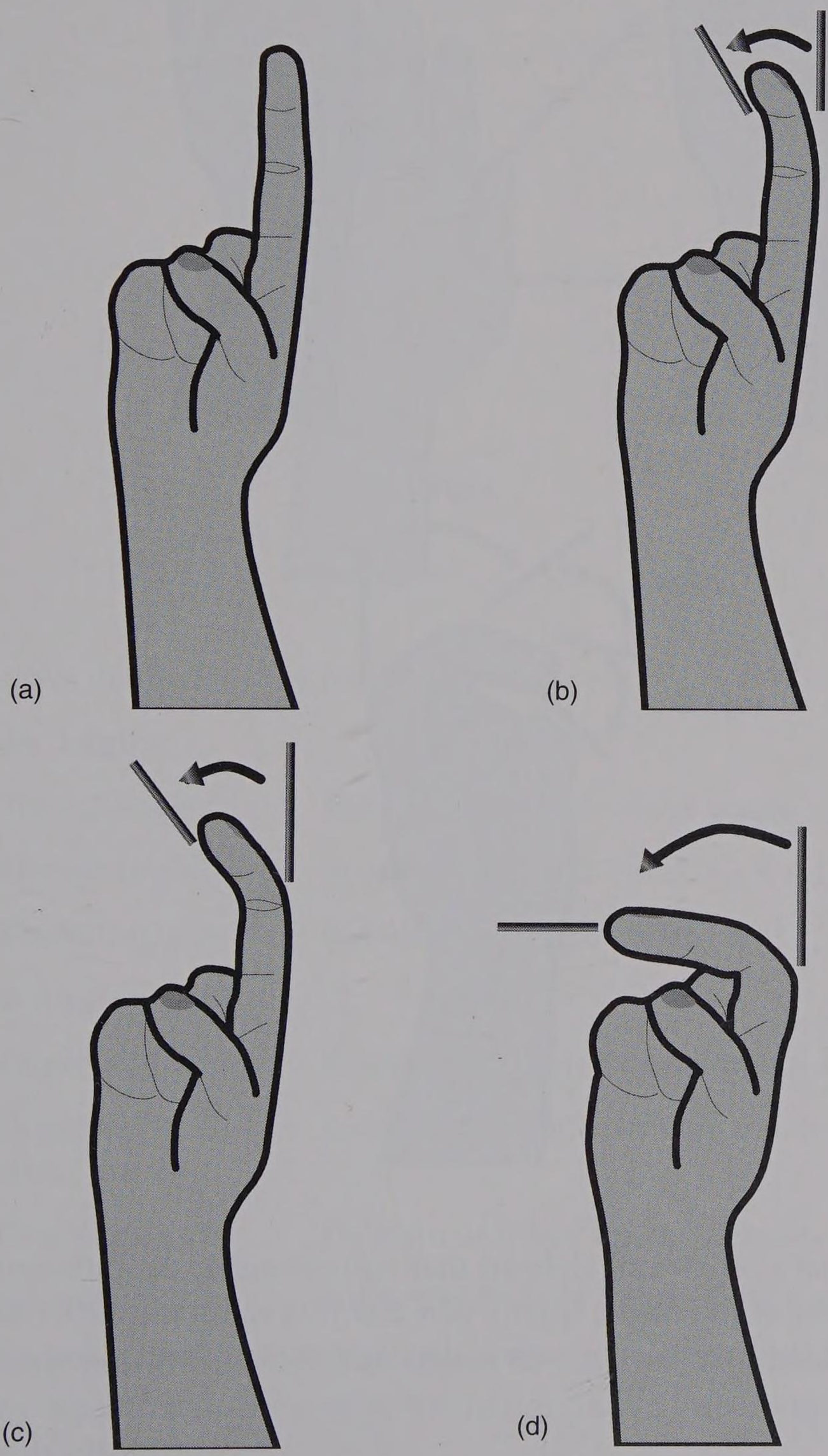
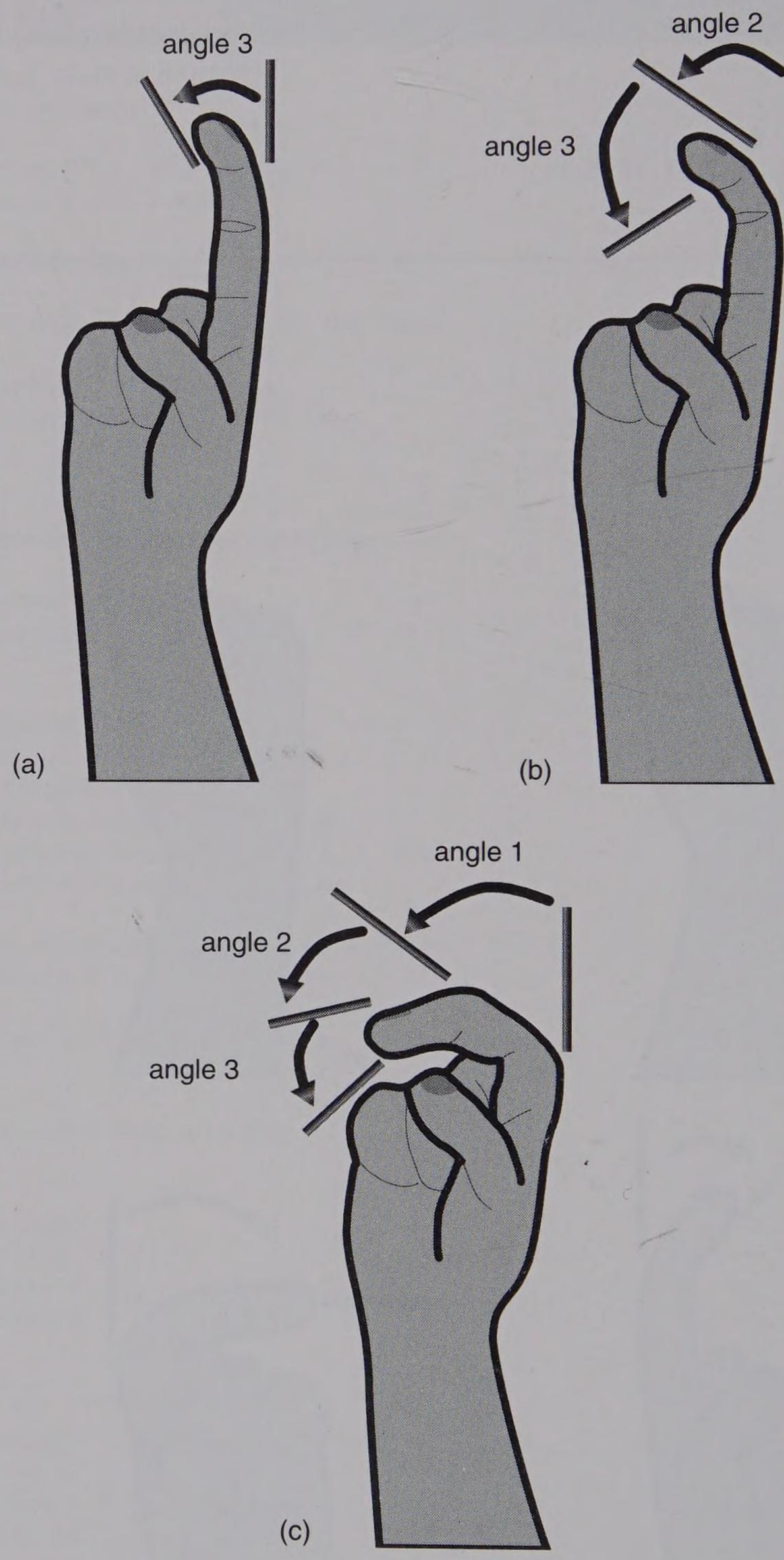


Figure 8.14

Bending different sections of a finger.

Figure 8.15

The relationship between angles in the fingers.



In HAND.PI, a set of angles are defined that correspond to the different angles in the fingers and in the thumb (the thumb's motion is different from the rest of the fingers). For orientation purposes, we will assume that

the palm of the open right hand is facing the -z axis and the fingers, when stretched, are pointing in the +y axis. When the thumb is stretched out completely, it points in the +x axis (see fig. 8.16).

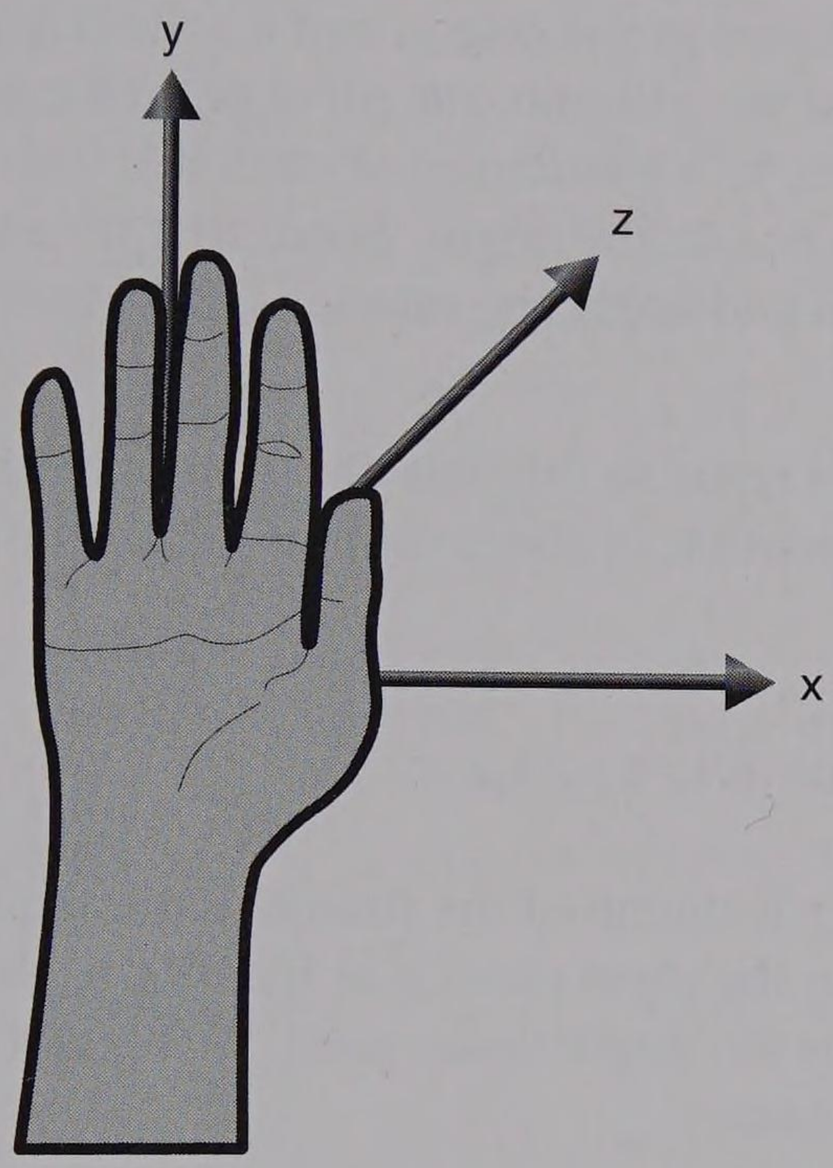


Figure 8.16
Coordinate system for the hand model.

The angles defined for the hand are

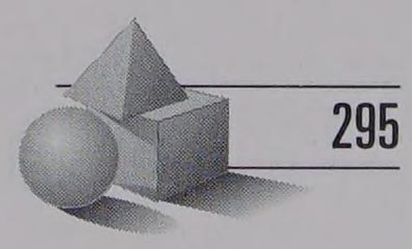
Thumb Angles:

ThumbAngley	Rotation of thumb about y axis
ThumbAngle2x	Rotation of thumb tip about x axis
ThumbAnglez	Rotation of thumb about z axis

Finger Angles:

FingerAnglex123	Rotation of whole finger about x axis
FingerAnglex23	Rotation of middle section and tip of finger about x axis
FingerAnglex3	Rotation of finger tip about x axis
FingerAnglez	Rotation of whole finger about z axis

Let's study how the thumb is modeled. The thumb is composed of two sections: tip and base (nearest to the hand). Start by defining a generic piece of thumb with a box:



```
define ThumbPart
object {
    box < -1, -1, -1 >, < 1, 1, 1 >
    scale < 0.3,0.5,0.5 >
}
```

The unit box is located in the origin, and a scale factor of <0.3, 0.5, 0.5> is used. If you look at your thumb, the pivot point for each section is at the base of that section, so a translation of <0.0, 0.5, 0.0> is required to locate the bottom of the box in the origin. Recall that it's easier to locate the pivot point in the origin and apply the rotations, then translate it to its final location.

The section `THUMB 1` refers to the base of the thumb. The only transformation applied is the translation of the pivot point to the origin.

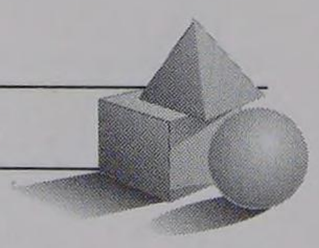
```
// BOX THUMB 1
    object {
        ThumbPart
        translate< 0.0, 0.5, 0.0 >
    }
```

The section `THUMB 2` is the tip of the thumb and it requires more work. First, translate the tip so the pivot point is at the origin. Next, apply a rotation about the x axis by the angle `ThumbAngle2x`. Finally, locate the tip of the thumb on top of `THUMB 1`.

```
// BOX THUMB 2
    object {
        ThumbPart
        translate< 0.0, 0.5, 0.0 >
        rotate< ThumbAngle2x, 0.0, 0.0 >
        translate< 0.0, 1.0, 0.0 >
    }
```

After constructing each section of the thumb, you still have some more transformations to apply. At this point, the pivot point of the section `THUMB 1` is located at the origin, and `THUMB 2` is on top of `THUMB 1`. When the thumb rotates about z, the whole thumb rotates. The transformation `rotate < 0.0, 0.0, ThumbAnglez>` is applied to the whole thumb. The next transformation `rotate < 0.0, ThumbAngley, 0.0>` is applied also to the whole thumb but it is done after the z rotation. To experiment, try swapping these to operations. You'll see that the thumb will rotate in a way that is not physically possible. After all the rotations have been applied, its time to move the complete thumb to its place in the hand. The transformation `translate < 1.75,0.0,0.0>` does this.

The same technique is used to model the finger. Since the finger is divided into three sections, modeling it is more involved, but still straightforward.



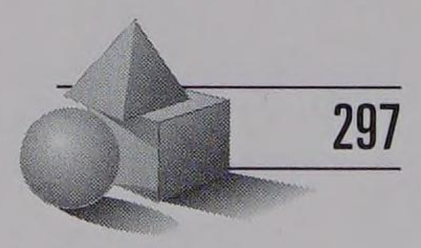
After one finger has been modeled, you can create multiple copies of it and locate them in their respective places in the hand:

```
// INDEX
object {
   WholeFinger
   scale < 0.9, 0.8, 1 >
   translate < 1.1,1.75,0.0 >
   HandTexture
}

// MIDDLE
object {
   WholeFinger
   scale < 0.9, 1, 1 >
   translate < 0.3,1.75,0.0 >
   HandTexture
}

// RING
object {
   WholeFinger
   scale < 0.9, 0.9, 1 >
   translate < -0.5,1.75,0.0 >
   HandTexture
}

// PINKY
object {
   WholeFinger
   scale < 0.9, 0.75, 1 >
   translate < -1.3,1.75,0.0 >
   HandTexture
}
```

Notice that since the fingers of a hand have all different lengths, a scaling operation is used on each finger to change its length. This is by no means an accurate model of a hand, but it's a good foundation from where to start more complicated hands.

The next step is to animate the hand. To make things more interesting, halfway through the animation, a ball comes bouncing from nowhere and gets caught by the robot hand. The section of the code that sets up the animation for the robot hand is

```
start_frame 0
end_frame 45

define ThumbAngley    frame*2
define ThumbAngle2x   -frame
define ThumbAnglez    -frame

define FingerAnglex123 -frame
define FingerAnglex23 -frame
define FingerAnglex3 -frame
define FingerAnglez 0
```



The animation will have a total of 46 frames (0 to 45). Each angle will be equal to the frame number except `ThumbAngley` which will always be twice the frame amount `(frame * 2)`.

To animate the ball, I used POVCAD and created a set of points on the screen and saved them with the `FREEHAND` command (in the Path menu). These points describe the path of the ball. Then I used the program ARRAY.BAS to convert the points into an array named `BallPath`. There are only 15 points, so the motion of the ball starts sometime in the middle of the animation. The idea was to show the ball approaching the robot hand, first by the ball's shadow, then by its reflection in the palm of the robot hand, then by showing it as it enters the observer's field of view.

The ball starts the motion by means of a conditional processing expression:

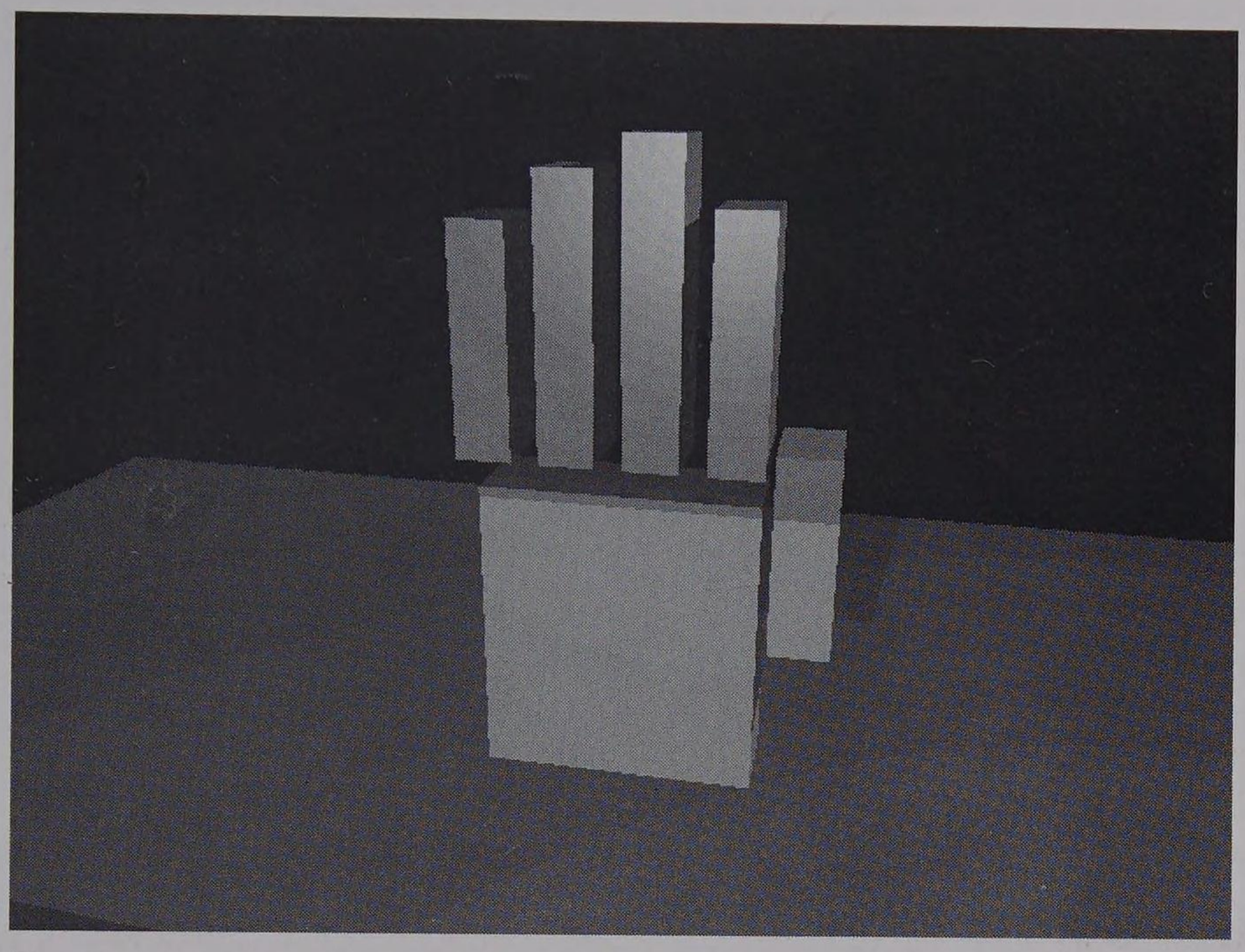
```
if (frame > 30)
object {
   sphere BallPath[frame - 31], 1
   shiny_red
}
```

If `frame` is greater than 30, the ball starts its motion. The location of the sphere is taken out of the array `BallPath`. To select the correct location, the expression `frame - 31` is used. When frame is equal to 31, then `frame - 31 = 31 - 31 = 0` (recall that the first element in the array has the index 0). When frame is 45, then `45 - 31 = 14` (the last element's index is 14).

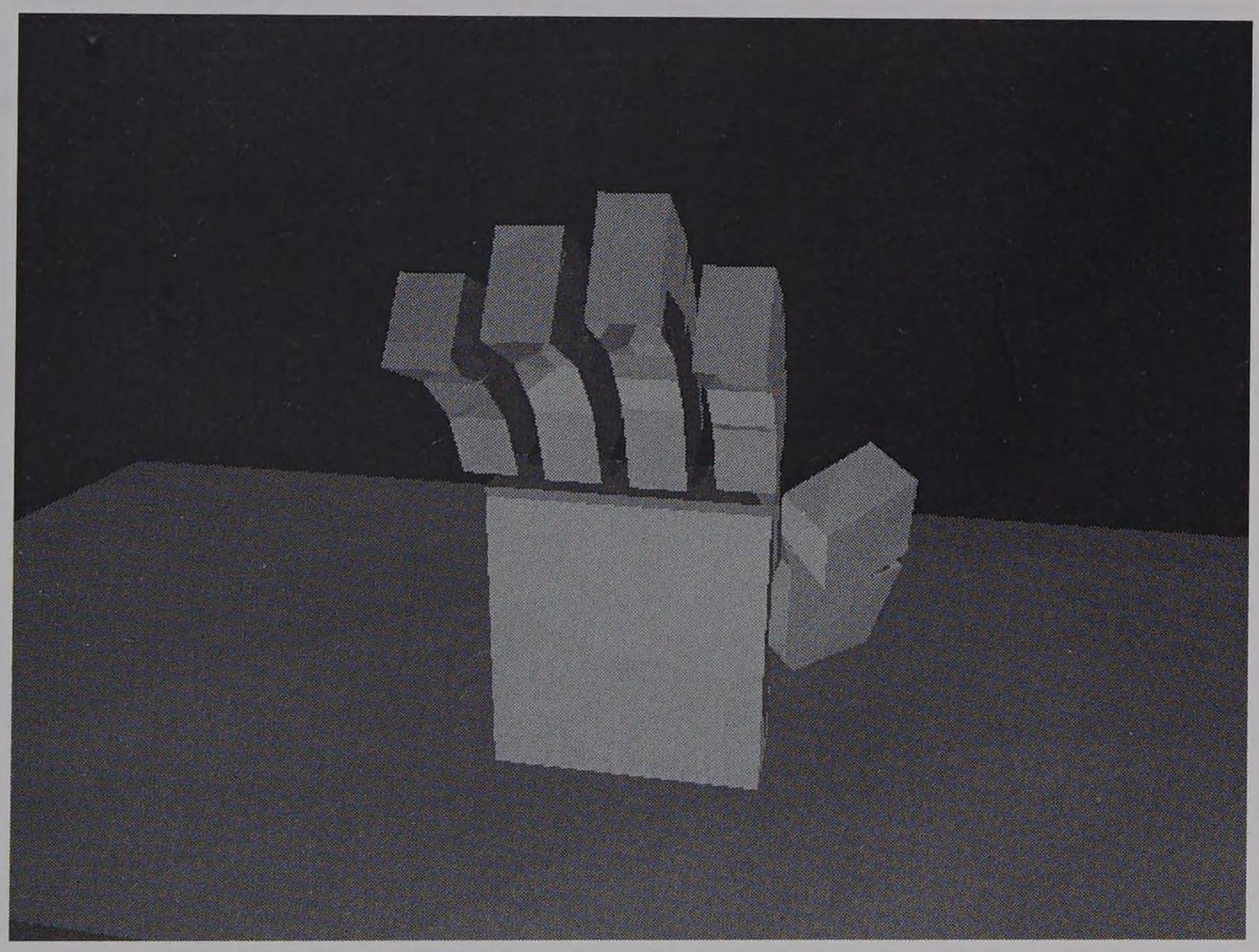
Figure 8.17 displays various frames of the animation.

Virtual Animation

As I mentioned earlier, in a virtual animation scene the object is shown in all of the locations at one time. Transparency is used as a means of distinguishing which is the sequence of the animation: the object in the first location will be the most transparent and the last object in the sequence will be totally opaque. Listing 8.8 presents a virtual animation scene.

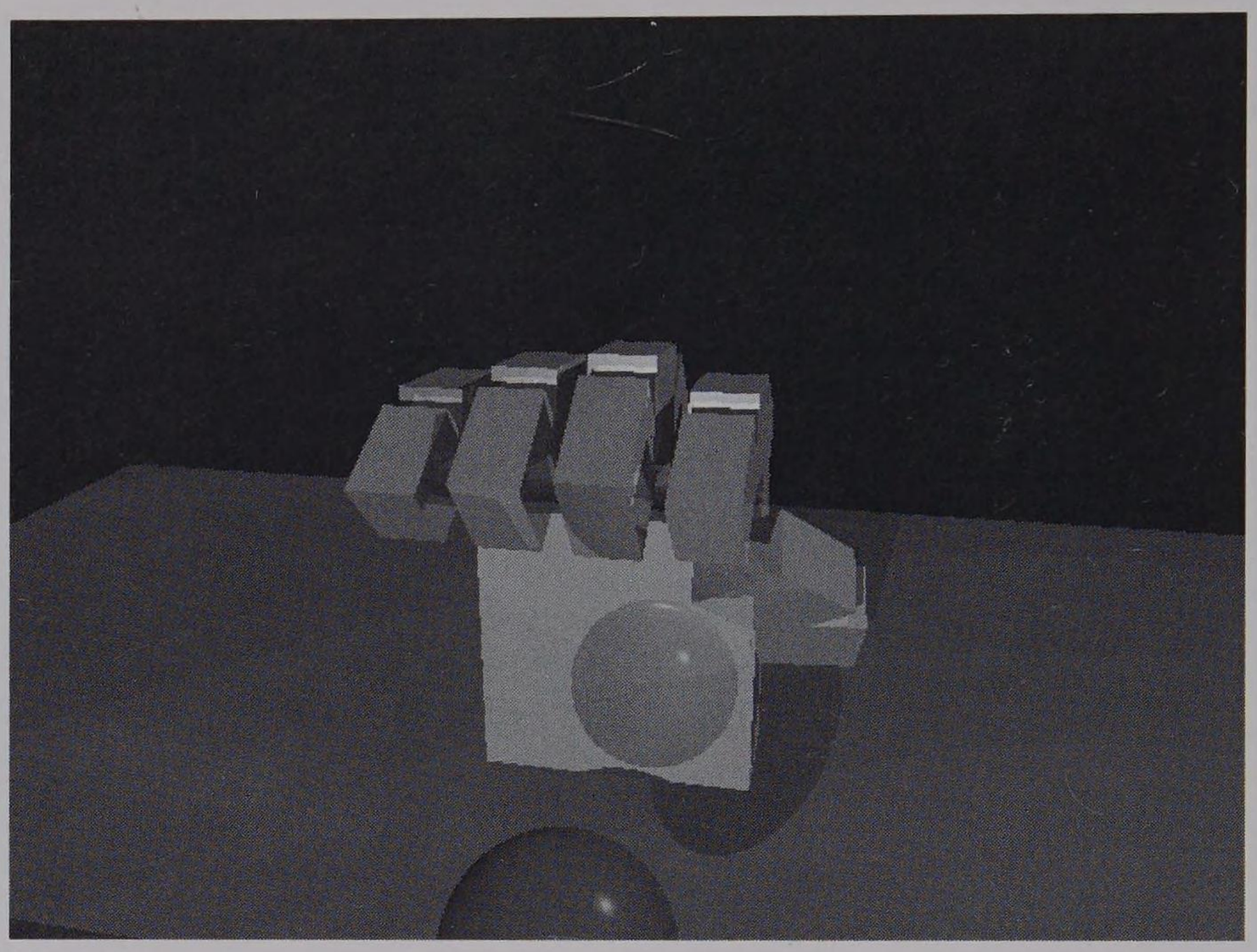


Frame 1

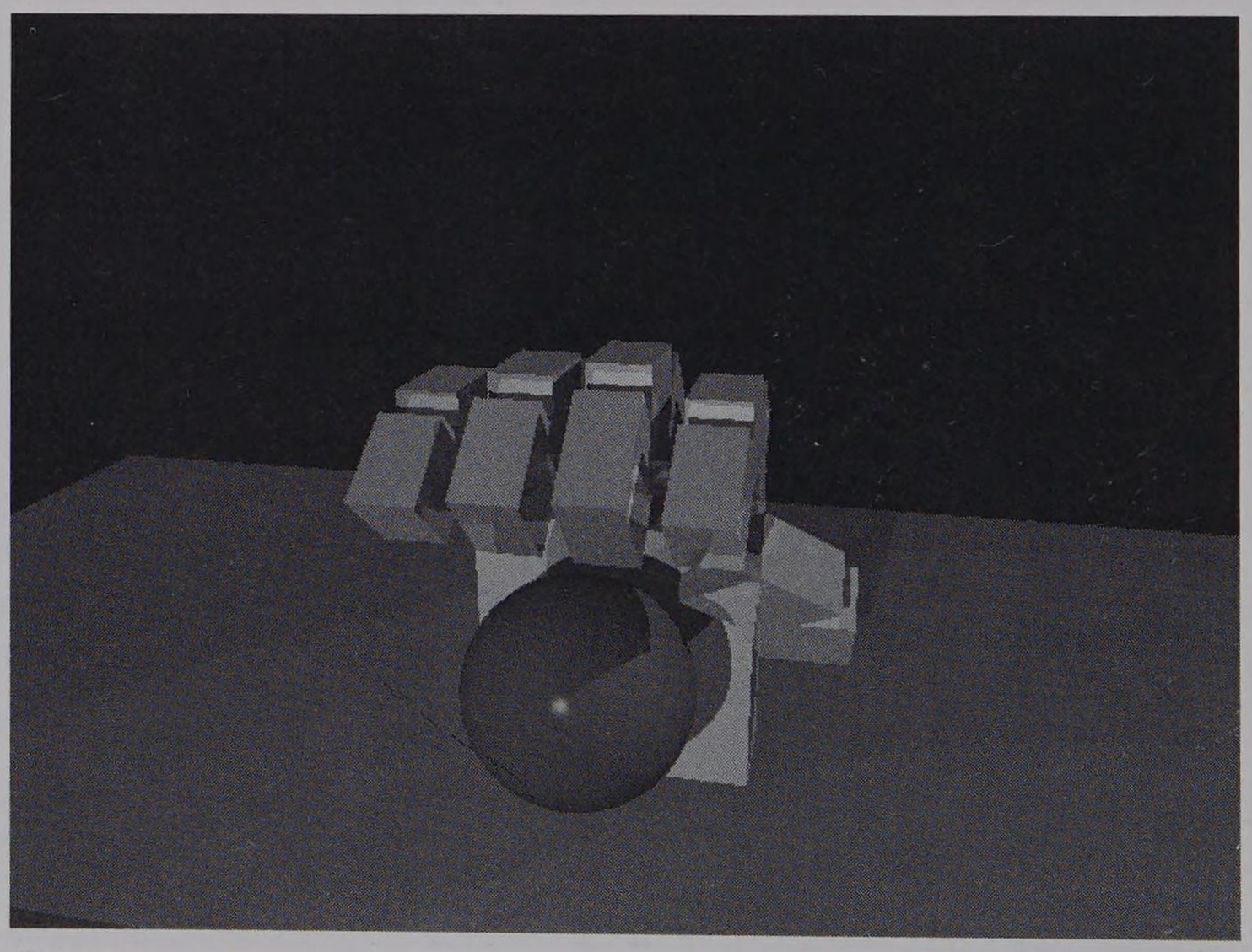


Frame 2

Figure 8.17
Robot hand animation (HAND.PI).



Frame 3



Frame 4

The scene VIRTUAL.PI has some interesting features. First, a set of textures *Tn* where *n* is a number indicating the location number, are defined:

```
define Transparent
   surface {
      color red
      ambient 0.1
      diffuse 0.7
      specular 0.2
      reflection white, 0.1
   }

define T1  texture {Transparent { transmission white, 1.00, 1}}
define T2  texture {Transparent { transmission white, 0.98, 1}}
define T3  texture {Transparent { transmission white, 0.96, 1}}
...etc...

define T19 texture {Transparent { transmission white, 0.00, 1}}
```

Since there are 19 spheres, the best way to reduce the amount of typing is by creating the surface definition "Transparent." Then each individual sphere will have its own transmission value.

The second feature is the use of an array to define the sphere locations. You saw this technique in the previous section where the robot hand catches a ball. This also considerably reduces the amount of typing. Figure 8.18 displays the final image.

Listing 8.8. Virtual animation of a bouncing sphere (VIRTUAL.PI).

```
// VIRTUAL.PI
include "colors.inc"
include "texture.inc"
background grey
define Transparent
   surface {
      color red
      ambient 0.1
      diffuse 0.7
      specular 0.2
      reflection white, 0.1
   }
define T1  texture {Transparent { transmission white, 1.00, 1}}
define T2  texture {Transparent { transmission white, 0.98, 1}}
define T3  texture {Transparent { transmission white, 0.96, 1}}
define T4  texture {Transparent { transmission white, 0.94, 1}}
define T5  texture {Transparent { transmission white, 0.92, 1}}
define T6  texture {Transparent { transmission white, 0.90, 1}}
define T7  texture {Transparent { transmission white, 0.88, 1}}
define T8  texture {Transparent { transmission white, 0.86, 1}}
define T9  texture {Transparent { transmission white, 0.84, 1}}
define T10 texture {Transparent { transmission white, 0.82, 1}}
define T11 texture {Transparent { transmission white, 0.80, 1}}
define T12 texture {Transparent { transmission white, 0.78, 1}}
define T13 texture {Transparent { transmission white, 0.76, 1}}
```

continues

Listing 8.8. Continued.

```
define T14 texture {Transparent { transmission white, 0.74, 1}}
define T15 texture {Transparent { transmission white, 0.72, 1}}
define T16 texture {Transparent { transmission white, 0.70, 1}}
define T17 texture {Transparent { transmission white, 0.68, 1}}
define T18 texture {Transparent { transmission white, 0.66, 1}}
define T19 texture {Transparent { transmission white, 0.00, 1}}
define BallPath [
                <-5.9184,  1.8992,  0>,
                <-5.5152,  2.3744,  0>,
                <-4.7856,  2.6304,  0>,
                <-4.1648,  2.448 ,  0>,
                <-3.77   ,  2.05  ,  0>,
                <-3.36   ,  1.3152,  0>,
                <-3.032  ,  0.4384,  0>,
                <-2.6    ,  0.95  ,  0>,
                <-2.0096,  1.5344,  0>,
                <-1.4608,  1.7904,  0>,
                <-0.8032,  1.7904,  0>,
                <-0.2928,  1.2784,  0>,
                < 0.0736,  0.74  ,  0>,
                < 0.4384,  0.4016,  0>,
                < 1.1696,  0.9136,  0>,
                < 1.7168,  1.2416,  0>,
                < 2.3744,  1.2048,  0>,
                < 2.9216,  0.9136,  0>,
                < 3.3968,  0.4016,  0>
          ]

define Plane_XZ
object { polygon 4, <-20,0,20>, <-20,0,-20>, <20,0,-20>, <20,0,20> }
// LIGHT_SOURCE
light < 10 ,  10 , 0  >
// LIGHT_SOURCE
light < -1 ,  5 , -20  >
viewpoint {
 from      <0.0, 2.0, -15.0 >
 at        <-1.0, 1.5, 0.0 >
 up        <0.0, 1.0, 0.0 >
 angle      40
 resolution 200,200
 aspect     1.0
}
// FLOOR
object {
     Plane_XZ
     matte_brown
}

object {
   sphere BallPath[0], 0.4
   T1
}
object {
   sphere BallPath[1], 0.4
   T2
```

```
}
object {
   sphere BallPath[2], 0.4
   T3
}
object {
   sphere BallPath[3], 0.4
   T4
}
object {
   sphere BallPath[4], 0.4
   T5
}
object {
   sphere BallPath[5], 0.4
   T6
}
object {
   sphere BallPath[6], 0.4
   T7
}
object {
   sphere BallPath[7], 0.4
   T8
}
object {
   sphere BallPath[8], 0.4
   T9
}
object {
   sphere BallPath[9], 0.4
   T10
}
object {
   sphere BallPath[10], 0.4
   T11
}
object {
   sphere BallPath[11], 0.4
   T12
}
object {
   sphere BallPath[12], 0.4
   T13
}
object {
   sphere BallPath[13], 0.4
   T14
}
object {
   sphere BallPath[14], 0.4
   T15
}
object {
   sphere BallPath[15], 0.4
   T16
```

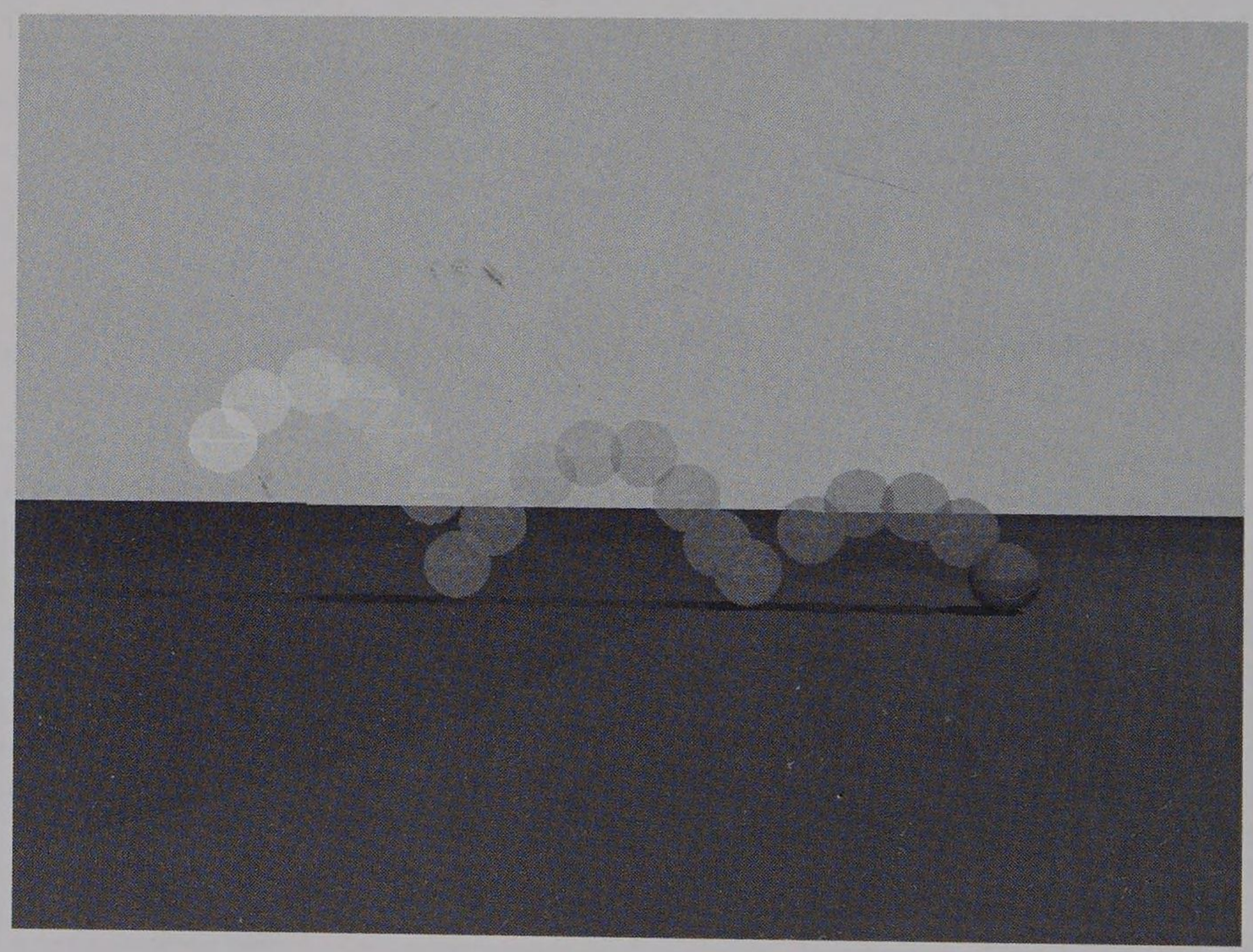
continues

Listing 8.8. Continued.

```
}
object {
   sphere BallPath[16], 0.4
   T17
}
object {
   sphere BallPath[17], 0.4
   T18
}
object {
   sphere BallPath[18], 0.4
   T19
}
```

Figure 8.18

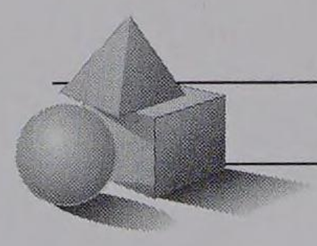
Virtual animation of a bouncing ball.



Summary

In this chapter, you studied different types of animation: by moving the viewpoint and the "look at" you can create the effect of a flyby. Textures can be animated by applying transformation in an object after the texture has been defined for it. Another type of animation called "virtual animation" is used to produce a static scene that would represent how an object moves in a sequence. This is accomplished by creating multiple copies of the object as it travels from one location to another. The effect is completed by varying the amount of transparency in the object's surface.

Now that you're done with the book, where do you go from here? My suggestion to you is this: take some time to experiment changing different parameters in the scenes included with this book. This teaches you how small changes in the parameters affect the final image. In terms of animation, try modeling different objects: a pair of scissors, a spinning top, a person walking, and so on. Some of these aren't easy, but you can model them by carefully observing which features are the most outstanding and then modeling those to give the animation the effect of reality. Finally, go to your nearest bookstore and get a couple of books on animation. I suggest the book *Advanced Animation and Rendering Techniques*, by Watt and Watt, as an excellent reference.

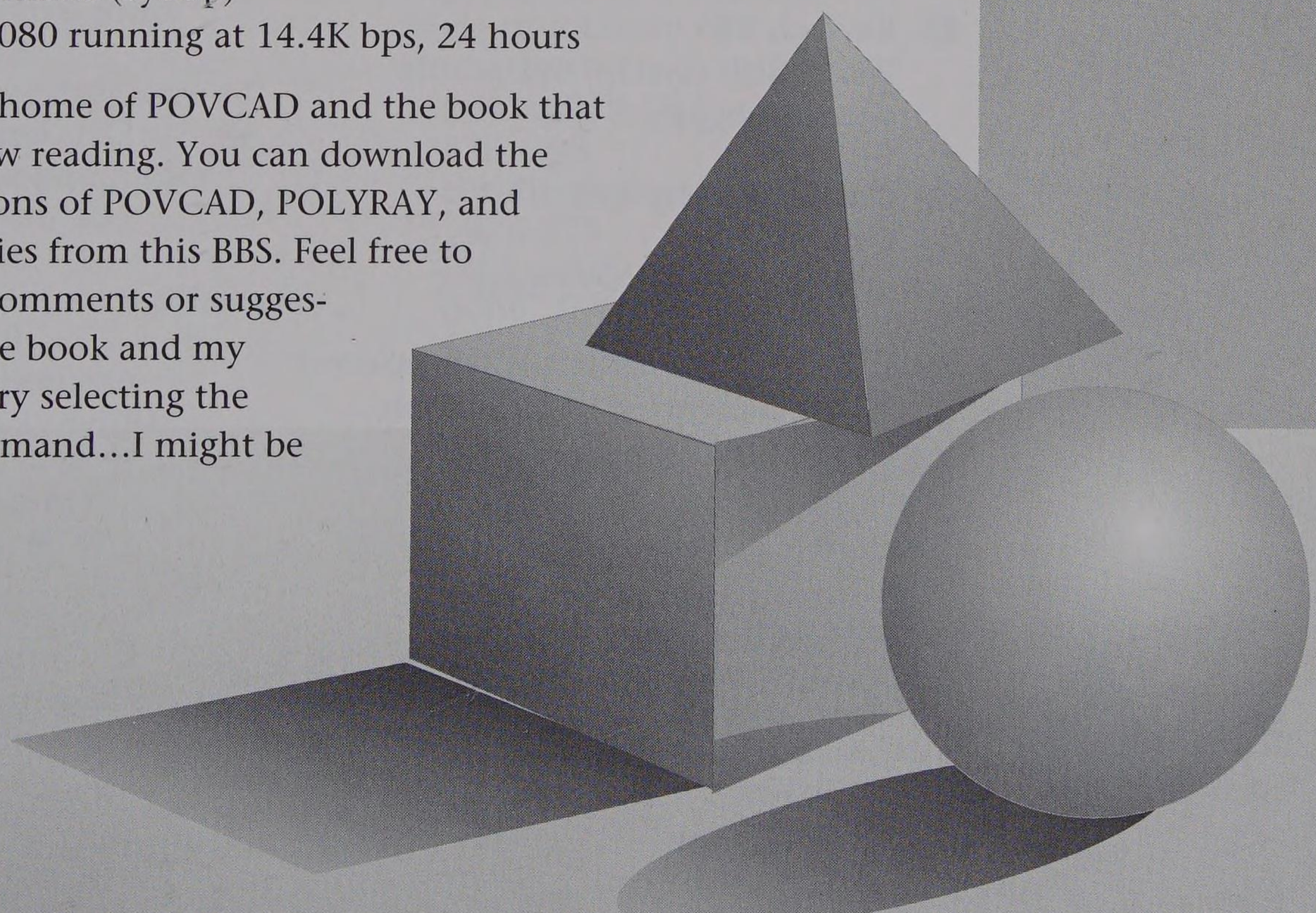


Where Can I Find Ray Tracing Software?

Here's a list of bulletin boards where you can find the latest and greatest in ray tracing software and utilities:

- Pi Squared BBS (Maryland)
  Alfonso Hermida (SysOp)
  (301)725-9080 running at 14.4K bps, 24 hours

  This is the home of POVCAD and the book that you are now reading. You can download the latest versions of POVCAD, POLYRAY, and other utilities from this BBS. Feel free to leave any comments or suggestions on the book and my program. Try selecting the [C]hat command...I might be around!



- CompuServe
  (614)-457-8650
  (800)-848-8990
  GraphDev Forum
  Alfonso Hermida, ID# 72114,2060

  The GraphDev forum in CompuServe is *the* place where people interested in computer graphics development, ray tracing, fractals, animation, and image processing meet. There are various message areas depending on the topic of your interest and file areas with utilities to aid you. There is also an extensive image file area inside GraphDev full of GIF files for you to download (these image files are created by people like you who enjoy sharing them with others).

- You Can Call Me Ray BBS (Illinois)
  Bill Minus/Aaron Collins (SysOps)
  (708)358-5611

- The Graphics Alternative BBS (California)
  Adam Shiffman (SysOp)
  (510)524-2780

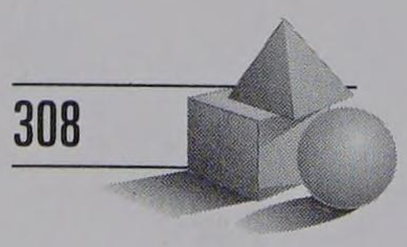
- Monsoon BBS (Maryland)
  Sunil Gupta (SysOp)
  (410)235-2365

In Europe:

- Raytech BBS
  Paul Smith (SysOp)
  +44 862 88340

- Digital Imagery BBS
  Philip Harris
  +44 295 272881 V32 MNP 5

- CIX (UK equivalent of CompuServe)
  ECS conference, POVCAD topic
  +44 081 390 1255



Index

Symbols

& (clipping) Boolean operation, 163, 173-174
* (asterisk), 23, 163, 171-172
+ (plus sign)
  Boolean operation, 163-168
  CSG (constructive solid geometry), 23
- (minus sign)
  Boolean operation, 163, 168-170
  CSG (constructive solid geometry), 23
-q command line parameter (shading options), 182
-rl command line parameter (scan conversion), 184
// (comments), 36
{ } (curly brackets) in conditional expressions, 279
~ (tilde) inverse Boolean operation, 163, 178-180
3-D
  coordinate systems
    left-handed, 8, 47
    objects, 11
    origin, 10
    points, 10
    right-handed, 8
    rotations, 10
    vectors, 48
  geometry, 17-20
  objects
    coordinate location, 11
    modeling, 5
  see also POVCAD

A

ambient light, 122
angle (spot light source), 110
angle of rotation, 49
animation
  arrays, 279-281
  ball, 298
  conditional processing, 276-278
  DTA utility, 281
  flyby, 269-271, 282-287
  frame generation, 274-276
  motion path, 269
  objects, 290-298
  orbits, 286
  Polyray, 274-282
  resolution, 282
  robot hand, 290
  static definitions, 278-279
  textures, 273, 287-296
  transformations, 266-269

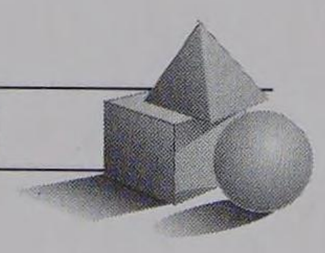
viewpoint motion, 269
virtual animation, 272, 298-304
animation files, 281-282
ANIMCLIP.PI (Listing 8.2), 275
antialiasing, 27, 31
apex point
cones, 40, 70
parabolas, 68
rotation operations, 49
spot light source, 110
apex radii, 40
apogee (elliptical orbit), 283
ARRAY.BAS (Listing 8.3), 281
ARRAY.OUT (Listing 8.4), 281
arrays
animation, 279-281
mathematical expressions, 194-195
aspect ratio, 38
attaching textures to objects, 208-209
attractive properties, 99
axes
3-D coordinate systems, 8
contour curves, 19
AXISORBT.PI (Listing 8.6), 288-290

B

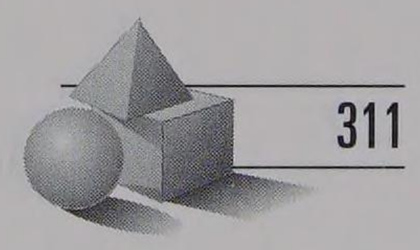
BACK.PI (Listing 4.4), 118
background image files, 117-118
ball (animated), 298
base point
cones, 40, 70
parabolas, 68
rotation operations, 49
base radii, 40
batch files, 42
BEZ file name extension, 250
Bézier objects, 250
Bézier patches, 94-98
color maps, 137
deforming, 252
POVCAD program, 20
POVCAD tutorial, 249-254
smooth surfaces, 20
Bézier submenu, 250
BEZIER.PI (Listing 3.19), 95-97
BLOB.PI (Listing 3.20), 99-100
blobs, 98-100
blue glass surface, 121
blurred images, 51
Boolean operations, 24-26
& (clipping), 163, 173-174
* (intersection), 163, 171-172
+ (union), 163-168
- (difference), 163, 168-170
~ (inverse), 163, 178-180
creating primitives, 229
boundaries, clipping, 173
BOUNDED.PI (Listing 5.5), 176-177
bounding boxes, 175-177
BOX (Listing 2.6), 39
boxes, 39-40, 54-55, 215
bounding, 175-177
intersecting with torus, 230
unit boxes, 55-56
BOXES.PI (Listing 3.3), 54-55
buffers, 241
building objects, 22-26
bulletin boards, 307-308
bump_scale keyword, 146

C

calculations
reflection values, 51
speeding with bounding boxes, 177
transparency, 51
vectors, 188
case-sensitivity, 113
center points (spheres), 59
centering, 213, 220-223
checkered finishes, 95
checkered texture, 132
CIX (on-line information service), 308
CLIP.PI (Listing 5.4), 173-174
clipping (&) Boolean operation, 163, 173-174



clipping bounding boxes, 176-177
CMAPPER utility, 141
color, 113-120
  background, 117-118
  haze, 118-120
  light sources, 112
  modeling, 6
  noise function, 193
  Polyray, 117-118
  predefined, 113
  scale, 122
  spot light, 110-112
  surfaces, 116-117
  textures, 117, 122
  turbulence, 192
  vectors, 113
color maps, 136-141
  Bézier patches, 137
  functional textures, 155
color vectors, 106
color wheel, 191-195
COLORMAP.PI (Listing 4.7), 138-141
COLORS.DOC (Polyray), 114-116
command-line parameters, 31-34
  -q (shading options), 182
  -r1 (scan conversion), 184
  antialiasing, 31
  batch files, 42
  file output operations, 32
  image quality, 32-33
  miscellaneous, 34-45
commands
  File menu
    Export, 210
    Save As, 210
  Misc menu, Viewpoint, 209
  Object menu
    Plane_Y, 214
    Sphere, 207
    Sweep, 78
comments (//), 36-37
complex objects, 22-26
components (vectors), 188
composite objects, 165-168
CompuServe, 308
COND.PI (Listing 6.5), 197-198
conditional expressions, 195-199
  { } (curly brackets), 277
  nesting, 277
  static definitions, 278
conditional processing (animation), 276-278
CONE (Listing 2.7), 40
cones, 40, 70, 216
constructive solid geometry, *see* CSG
contour curves
  lathe surfaces, 19
  solid of revolution, 86
conventions (3-D coordinate systems), 8
converting polygons to triangles, 79-86
coordinate systems, 47-48
  global, 142
  hand model, 295
  left-handed, 47
  local, 141
  noise surface texture, 141-142
  vectors, 48
  see also 3-D, coordinate systems
coordinates (boxes), 54
copying objects, 220-229
corner points (boxes), 54
CSG (constructive solid geometry)
  + (plus sign) operator, 23
  - (minus sign) operator, 23
  Boolean operations, 24-26
  intersections, 23
  menu, 231
  POVCAD tutorial, 229-239
  union operations, 23
current directory, 43
curves, *see* contour curves
CYLINDER.PI (Listing 3.12), 70-71
cylinders, 70-72



D

data files, 242, 280
databases, 30
declaring textures, 42, 273
decomposing polygons, 255-263
define keyword, 52, 55-56
defining
  color textures, 117
  images, 38
  light source, 38
  objects, 38-41
  parabolas, 68
  planes, 56-72
  primitives, 227-230
  vectors, 190
  viewpoints, 37-45, 50-52, 209
definitions, 52-53
deleting points, 241
delta (elliptical orbit), 283
DEMO.PI Listings (2.1 through 2.4), 36-38
depth of field, 51
DIFFR.PI (Listing 5.2), 169
diffuse light, 122
Digital Imagery BBS, 310
directional light sources, 106-109
DIRECTLT.PI (Listing 4.2), 108-109
directories, 43
discs
  position (Listing 3.9), 66
  radius, 65
  simple objects, 65-68
DISCS.PI (Listing 3.10), 66-67
distortions, 143
donut shapes, *see* torus
DOS (disk operating system), 203
DRAFT.BAT file, 42
DTA utility, 281

E

EDIT editor, 35
Elbow primitive, 229-239
ellipsoid, 215
  deforming, 250
  scaling spheres, 72
elliptical orbit, 283
END statement, 230
endpoints, 70
exiting POVCAD, 210
Export command (File menu), 210
exporting to Polyray files, 218
expressions
  conditional expressions
    { } (curly brackets), 277
    nesting, 277
    static defintions, 278-279
  mathematical, 185-198
    arrays, 194-195
    conditional expressions, 195-199
    floating-point, 185-188
    vectors, 188-194
extrude objects (POVCAD tutorial), 240-249
extruded surfaces, 20, 91-94

F

fall-off spot light source, 110
Figures in Polyray format (Listing 8.1), 268-269
File menu commands
  Export, 210
  Save As, 210
file output operations, 32
files
  animation, 281-282
  batch, 42
  data, 242
  path, 242
  POLYRAY.EXE, 30
  saving, 210
  scene, 34-42
    creating, 207-213
    Polyray, 212-213
  Targa, 43
flags, 182-184
flat triangles, 76, 81
FLAT.BAS (Listing 3.14), 79-80
floating point expressions, 185-188

flyby animation, 282-287
formats
  POVCAD, 210-211
  RAW data format, 78
Fractint, 101
frames, 274-276
frequency keyword, 146
functional textures, 131, 154-157
  color map, 155
  lookup functions, 155
functions, 103-104
  floating point number return, 186-187
  integer return, 186-187
  lookup functions, 155
  mathematical, *see* mathematical functions
  vector return, 189-190

G

generating frames (animation), 274-276
geometry (3D), 17-20
global coordinate system, 142
GraphDev forum (CompuServe), 308
Graphics Alternative BBS, 308
grid, 221
grouping
  expressions into arrays, 194
  objects, 163
groups, 164

H

HAND.PI (Listing 8.7), 290-293
haze, 118-120
header section
  DEMO.PI, 36-37
  PLANES.PI (Listing 3.6), 59
height fields, 101-102
HEIGHT.PI
  Listing 4.5, 119-120
  Listing 3.21, 102
HEX.PI (Listing 4.6), 134-135
hexagon texture, 132-135

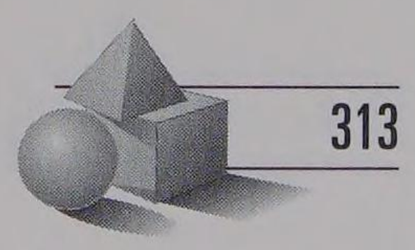
I

icons, 205-207
image files
  background, 117-121
  height fields, 101
image mapping textures, 158-162
images
  aspect ratio, 38
  blurred, 51
  quality (Polyray), 32-33
  size, 38
  sizing, 43
  viewing, 43-44
include statements, 37
index of refraction (IOR), 125
infinite planes, 56
integers, 186-187
INTERS.PI (Listing 5.3), 171-172
intersection (*) Boolean operations, 163, 171-172
intersections
  CSG (constructive solid geometry), 23
  limiting, 51
  torus and box, 230
inverse (~) Boolean operation, 163, 178-180
INVERSE.PI (Listing 5.6), 178-179
IOR (index of refraction), 125

J-K

jaggies, 27
joining objects, 164-168
  see also unions

keywords
  background, 117
  bump_scale, 146
  define, 52, 55-56
  frequency, 146
  lookup_fn, 143-144
  normal, 145
  octaves, 143
  phase, 146
  position_fn, 141



position_scale, 142
shading flags, 183-184
turbulence, 143

L

landscapes, 101-102
lathe objects, 19, 86-91, 247
LATHE.BAS (Listing 3.16), 87-88
LATHE.PI (Listing 3.17), 89-91
layered textures, 131, 152
LAYERED.PI (Listing 4.11), 153
layering, 152-153
left-handed coordinate system, 8-10, 47
light
ambient, 122
diffuse, 122
index of refraction, 125
modeling, 5
properties, 5
reflected light, 125
specular highlights, 123-125
transmitted, 125-126
light sources, 14-15, 106-113
color, 112
creating, 209
defining, 38
directional, 106-109
flyby, 270-271
formats, 211
motion, 286
point source, 14
positional, 106
spot light, 14, 109-113
tutorial, 216-217
LIGHT.PI (Listing 4.1), 107
limiting intersections, 51
limits (bounding boxes), 175-177
local coordinate system, 141
lookup functions, 155
lookup_fn keyword, 143-144

M

major radius (torus), 72
mapping
color, 136-141
textures, 27
MAPS.PI (Listing 4.13), 159-161
mathematical expressions, 185-198
arrays, 194-195
conditional expressions, 195-199
floating-point, 185-188
vectors, 188-194
mathematical functions, 30
menus
Bézier submenu, 250
CSG menu, 231
microfacet distribution, 126-129
minor radius (torus), 72
Misc menu commands, Viewpoint, 209
miscellaneous command-line parameters, 34-45
miscellaneous objects, 53, 98-104
modeling
3-D objects, 5
color, 6
light, 5
surfaces, 5, 76-98
Bézier patches, 53, 94-98
extruded surfaces, 91-94
lathe objects, 86-91
polygons, 86
sweep surfaces, 91-94
triangles, 53
textures, 6
models
microfacet models, 128
standard shading model, 121-131
Monsoon BBS, 308
MOON.PI (Listing 4.8), 146-147
motion
flyby, 269-271
light sources, 286

objects, 271-272
viewpoints, 269-271
see also animation
motion path, 269
moving light sources, 270
MTV ray tracer, 30

N

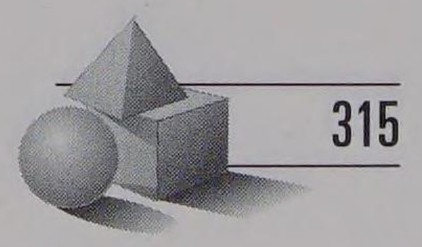
naming definitions, 52-53
negative rotation (3-D coordinate system), 10
nesting
Boolean operations, 24
conditional expressions, 277
noise surface texture, 135-146
bump_scale keyword, 146
color mapping, 136-141
coordinate system, 141-142
frequency keyword, 146
lookup_fn keyword, 143-144
normal keyword, 145
octaves keyword, 143
phase keyword, 146
position_fn keyword, 141
position_scale keyword, 142
turbulence keyword, 143
normal keyword, 145
normal modifiers, 146-151

O

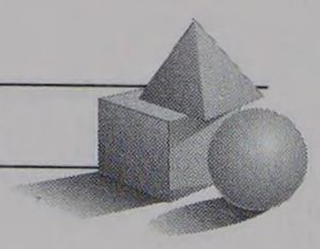
Object menu commands
Plane_Y, 214
Sphere, 207
Sweep, 78
objects
3-D, *see* 3-D objects
animation, 290-298
building, 22-26
composite, 165-168
copying, 220-229
creating, 30, 53
defining, 38-41
differencing, 168-170
lathe, 86-91
miscellaneous, *see* miscellaneous objects
motion, 271-272
patterns, 39
simple, *see* simple objects
textures, 208
unions, 165-168
windows, 196
observers, 14
see also viewpoint
octaves keyword, 143
optical properties of light, 5
ORBIT.PI (Listing 8.5), 284-285
orbital rotation, 65
order of transformations, 266
origin (rotations), 266-268
origin coordinates, 10
outlines, 77
over sampling, 27

P

PARABOLA.PI (Listing 3.11), 69
parabolas, 68-70
apex point, 68
base point, 68
defining, 68
rotating, 68
parameters
command-line parameters
-q (shading options), 182
-r1 (scan conversion), 184
antialiasing, 31
batch files, 42
file output operations, 32
image quality, 32-33
miscellaneous, 34-45
surfaces, 121-131
textures, 42, 129-131
viewpoints, 50-51
patches (contour curves), 20
path file, 242
patterns, 39, 132
perigree (elliptical orbit), 283
perturbation techniques, 27



phase keyword, 146
photo-realism, 26-28
photo-realistic images, 5
Pi Squared BBS, 307
pivot point, 265, 296
PLANE (Listing 2.5), 39
Plane_Y command (Object menu), 214
planes, 38-39, 56-59, 214
  bounding boxes, 175
  defining, 56-72
  infinite, 56
PLANES.PI (Listing 3.6), 57-58
point buffer, 241
point source, 14
points
  3-D coordinate systems, 10
  apex point
    cones, 40, 70
    parabolas, 68
    rotation operations, 49
  base point
    cones, 40, 70
    parabolas, 68
    rotation operations, 49
  center point, 59
  corner points, 54
  deleting, 241
  endpoints, 70
  outlines, 77
  pivot point, 267
polygons
  converting to triangles, 79-86
  decomposition, 255-263
  modeling surfaces, 86
  patterns, 39
  scan conversion, 184
Polyray, 30-34
  animation, 274-282
  color, 113-120
    background color, 117-118
    haze, 118-120
    surfaces, 116-117
  command-line parameters, 31-34
  scene files, 34-42, 212-213
  standard shading model, 121-131
  syntax, 35
  textures, 131-157
POLYRAY.EXE file, 30
position_fn keyword, 141
position_scale keyword, 142
positional light sources, 106
positioning
  disks, 66
  light sources, 209
  objects, 48-49
    see also translations
positive rotation (3-D coordinate system), 10
POVCAD, 20, 201-207
  Bézier patches, 249-254
  copying objects, 220-229
  CSG operations, 229-239
  exiting, 210
  exporting to Polyray files, 218
  extrude objects, 240-249
  formats, 210-211
  icons, 205-207
  polygon decomposition, 255-263
  primitives, 214-220, 229
  scene files, 207-213
  sweep objects, 240-249
  triangles, 77
  tutorials
    Bézier patches, 249-254
    CSG operations, 229-239
    polygon decomposition, 255-263
    primitives, 214-220
    scene files, 207-213
predefined color, 113
primitives, 53
  boxes, 54-56
  creating, 229
  database, 30
  defining, 227-230
  miscellaneous objects, 53
  modeling surfaces, 53, 76-98
  POVCAD, 214-220
  simple objects, 53



procedural textures, 131
properties
  attractive, 99
  light, 5
  repulsive, 99

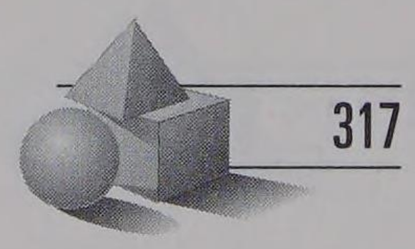
Q-R

QBasic, 79-84

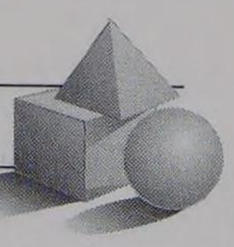
radius
  apex radii, 40
  base radii, 40
  cones, 70
  cylinders, 70
  discs, 65
  parabolas, 68
  torus, 72
RAW data format, 78
Raytech BBS, 308
reflected light, 125
reflective red surface, 121
rendering, 183-184
  scan conversion, 184
  wireframes, 184-185
repulsive properties, 99
resolution, 43, 282
right-handed coordinate systems, 8
ripples, 147
robot hand animation, 290-298
rotation, 49, 267
  angle of rotation, 49
  apex point, 49
  base point, 49
  groups, 164
  orbital rotation, 65
  parabolas, 68
  patterns, 132
  spheres, 65
  textures, 288-296
  torus, 226
rotations
  3-D coordinate systems, 10
  origin, 266-268
  pivot point, 267
running POVCAD, 203-207

S

Save As command (File menu), 210
saving files, 210
scale
  color, 122
  transmission, 130-131
scaling, 49
  box objects, 39
  image maps, 161
  spheres, 72
  tiles, 132
  see also sizing
scan conversion, 184
  see also wireframes
scene files, 34-42
  creating, 207-213
  header section, 36-37
  light source definition, 38
  object definition, 38-41
  Polyray, 212-213
  saving, 210
  text editor, 35
  textures, 41-42
  viewpoint definition, 37-38
scenes, 43
  reflection values, 51
  saving, 210
shading
  -q command line parameter, 182
  flags, 182-184
shareware, 30
simple objects
  boxes, 54-55
  cones, 70
  creating, 53-75
  cylinders, 70-72
  discs, 65-68
  ellipsoids, 72
  parabolas, 68-70
  planes, 56-59
  spheres, 59-65
  torus, 72-75
sizing images, 38, 43
  see also scaling
slabs (lathe surfaces), 19

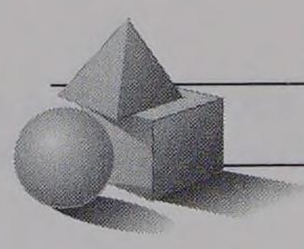


smooth surfaces, 20
smooth triangles, 76-84
SMOOTH.BAS (Listing 3.15), 81-84
smoothing, 184
snapping to grid, 221
software, 307
solids of revolution, 19, 86
SPECIAL.PI (Listing 4.12), 156-157
specular highlights, 123-125, 130
SPHERE (Listing 2.8), 41
Sphere command (Object menu), 207
spheres, 41, 59-65, 214-215
  center points, 59
  formats, 210
  rotation, 65
  scaling, 72
  textures, 208-209
SPHERES1.PI (Listing 3.7), 60-61
SPHERES2.PI (Listing 3.8), 62-64
spot light color, 14, 109-113
SPOTLITE.PI (Listing 4.3), 110-111
standard shading model, 121-131
static definitions, 278-280
subtracting objects (differencing), 168-170
surface finishes, 5
surfaces
  Bézier, 251
  blue glass, 121
  color, 116-117
  extruded, 20
  functions, 103-104
  intersections, 171-172
  lathe, 19
  modeling, 76-98
    Bézier patches, 94-98
    extruded, 91-94
    lathe objects, 86-91
    polygons, 86
    sweep, 91-94
  noise surface texture, 135-146
    bump_scale keyword, 146
    color mapping, 136-141
    coordinate system, 141-142
    frequency keyword, 146
    lookup_fn keyword, 143-144
    normal keyword, 145
    octaves keyword, 143
    phase keyword, 146
    position_fn keyword, 141
    position_scale keyword, 142
    turbulence keyword, 143
  normal modifiers, 146-151
  reflective red, 121
  ripples, 147
  smooth, 20
  sweep, 20
  texture
    blue glass, 121
    parameters, 121-131
  water, 147
Sweep command (Object menu), 78
sweep objects, 240-249
sweep surfaces, 20, 91-94
SWEEP.PI (Listing 3.18), 92-93
symmetric objects, 249
syntax
  ambient light, 122
  boxes, 54
  color maps, 136
  cones, 70
  define keyword, 52
  diffuse light, 122
  discs, 65
  frame creation, 274-275
  functional, 154
  haze, 118
  layered textures, 152
  microfacet models, 129
  noise surface, 135
  parabolas, 68
  Polyray, 35
  reflected light, 125
  surface color, 116
  translations, 48



T

Targa files, 43, 101
text editors, 35
texture models, 26
textures, 41-42, 121-162, 131-157
  ambient light, 122
  animation, 274, 287-296
  attaching to objects, 208-209
  checker, 132
  color, 122
  declaring, 275
  declaring parameters, 42
  defining color, 117
  diffuse light, 122
  functional, 131, 154-157
hexagon, 132-135
image mapping, 158-162
layered, 27, 131, 152-153
microfacet distribution, 126-129
modeling, 6
noise surface, 135-146
  bump_scale keyword, 146
  color mapping, 136-141
  coordinate system, 141-142
  frequency keyword, 146
  lookup_fn keyword, 143-144
  normal keyword, 145
  octaves keyword, 143
  phase keyword, 146
  position_fn keyword, 141
  position_scale keyword, 142
  turbulence keyword, 143
parameters, 129-131
perturbation techniques, 27
procedural textures, 131
reflected light, 125
reflective red surface, 121
specular highlights, 123-125, 130
surfaces, 121-131
tiles, 132
transmitted light, 125-126
thickness (haze), 118
tightness (spot light), 110
tiles
  hexagon texture, 132
  scaling, 132
torus, 72-75, 216
  centering, 220-223
  intersecting with box, 230
  rotating, 226
TORUS.PI (Listing 3.13), 73-75
transformations, 48-49, 266-269
  order, 266
  rotation, 49
  scaling, 49
  textures, 273
translations, 48-49, 161
transmission scale, 130-131
transmitted light, 125-126
transparency
  calculations, 51
  objects with windows, 197
  virtual animation, 272
transparent layers, 152
triangles, 53
  flat triangles, 76, 81
  RAW data format, 78
  smooth triangles, 76-84
  vectors, 76
  vertices, 76
turbulence, 192-195
turbulence keyword, 143
TUTOR2.PI (Listing 7.2), 218-219
TUTOR3.PI (Listing 7.3), 223-225
TUTOR3M.PI (Listing 7.4), 227-229
TUTOR4.PI (Listing 7.5), 235-236
TUTOR5.PI (Listing 7.7), 245-246
TUTOR6.PI (Listing 7.9), 252-254
TUTOR7.PI (Listing 7.10), 259
tutorials
  copying objects, 220-229
  POVCAD
    Bézier patches, 249-254
    creating scene file, 207-213
    CSG operations, 229-239
    extrude objects, 240-249
    polygon decomposition, 255-263
    primitives, 214-220
    Sweep objects, 240-249



U

union (+) Boolean operation, 163-168
union operations (CSG), 23
UNION.PI (Listing 5.1), 165-167
unions, 164-168
unit boxes, 55-56
utilities
  CMAPPER, 141
  DTA (animation), 281

V

vectors
  3-D coordinate system, 48
  color vectors, 106, 113
  components, 188
  defining, 190
  functions returning vectors, 189-190
  magnitude, 187
  mathematical expressions, 188-194
  triangles, 76
vertices (triangles), 76
viewing images, 43-44
viewpoint, 37-45
Viewpoint command (Misc menu), 209
viewpoints, 14, 50-52
  defining, 50-52, 209
  motion, *see* flyby
  parameters, 50-51
virtual animation, 272, 298-304
VIRTUAL.PI (Listing 8.8), 301-304

W

water surfaces, 147
WATER.PI (Listing 4.9), 148-149
WHEEL1.PI (Listing 6.2), 191
WHEEL2.PI (Listing 6.3), 192-194
Windows, 203
windows in objects, 196
wireframes, 184

X-Y-Z

You Can Call Me Ray BBS, 308

GO AHEAD. PLUG YOURSELF INTO PRENTICE HALL COMPUTER PUBLISHING.

Introducing the PHCP Forum on CompuServe®

Yes, it's true. Now, you can have CompuServe access to the same professional, friendly folks who have made computers easier for years. On the PHCP Forum, you'll find additional information on the topics covered by every PHCP imprint—including Que, Sams Publishing, New Riders Publishing, Alpha Books, Brady Books, Hayden Books, and Adobe Press. In addition, you'll be able to receive technical support and disk updates for the software produced by Que Software and Paramount Interactive, a division of the Paramount Technology Group. It's a great way to supplement the best information in the business.

WHAT CAN YOU DO ON THE PHCP FORUM?

Play an important role in the publishing process—and make our books better while you make your work easier:

- Leave messages and ask questions about PHCP books and software—you're guaranteed a response within 24 hours
- Download helpful tips and software to help you get the most out of your computer
- Contact authors of your favorite PHCP books through electronic mail
- Present your own book ideas
- Keep up to date on all the latest books available from each of PHCP's exciting imprints

JOIN NOW AND GET A FREE COMPUSERVE STARTER KIT!

To receive your free CompuServe Introductory Membership, call toll-free, **1-800-848-8199** and ask for representative **#K597**. The Starter Kit Includes:

- Personal ID number and password
- $15 credit on the system
- Subscription to CompuServe Magazine

HERE'S HOW TO PLUG INTO PHCP:

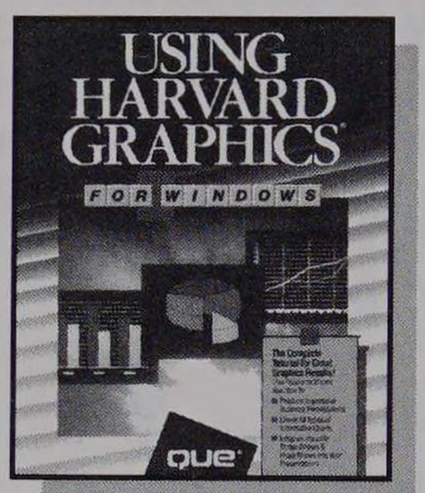
Once on the CompuServe System, type any of these phrases to access the PHCP Forum:

GO PHCP
GO QUEBOOKS
GO SAMS
GO NEWRIDERS
GO ALPHA
GO BRADY
GO HAYDEN
GO QUESOFT
GO PARAMOUNTINTER

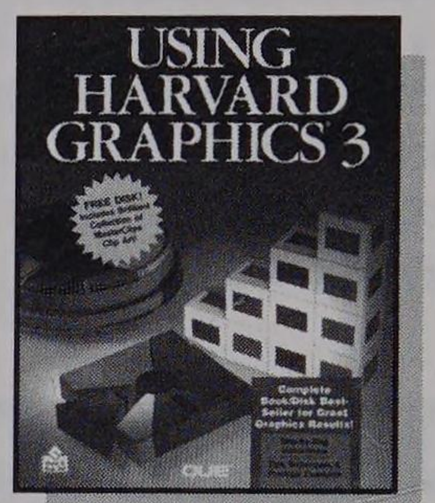
Once you're on the CompuServe Information Service, be sure to take advantage of all of CompuServe's resources. CompuServe is home to more than 1,700 products and services—plus it has over 1.5 million members worldwide. You'll find valuable online reference materials, travel and investor services, electronic mail, weather updates, leisure-time games and hassle-free shopping (no jam-packed parking lots or crowded stores).

Seek out the hundreds of other forums that populate CompuServe. Covering diverse topics such as pet care, rock music, cooking, and political issues, you're sure to find others with the same concerns as you—and expand your knowledge at the same time.

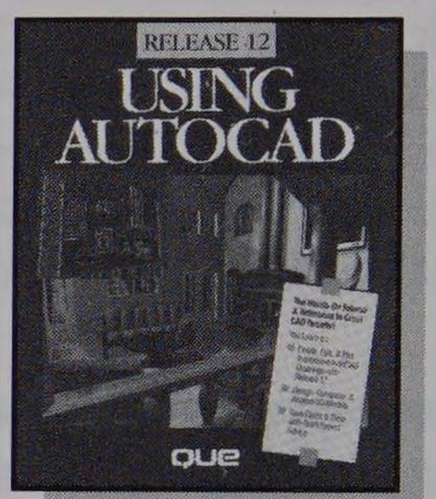
Let Que Help You with All Your Graphics Needs!



Using Harvard Graphics for Windows
Bob Benedict
Windows Version
$24.95 USA
0-88022-755-9, 555 pp., 7³/₄ x 9¼



Using Harvard Graphics 3
Steve Sagman & Jane Graver Sandlar
Version 3
$29.95 USA
0-88022-735-4, 700 pp., 7³/₄ x 9¼



Using AutoCAD Release 12
Que Development Group
Through Release 12
$34.95 USA
0-88022-941-1, 1200 pp., 7³/₈ x 9¼

More Graphics Titles from Que

AutoCAD Release 12 Quick Reference
Que Development Group
Releases 10, 11, & 12
$9.95 USA
1-56529-024-0, 160 pp., 4³/₄ x 8

Easy Harvard Graphics
Shelley O'Hara
Version 3
$19.95 USA
0-88022-942-X, 224 pp., 7³/₈ x 9¼

Harvard Graphics 3 Quick Reference
Trudi Reisner
Version 3
$9.95 USA
0-88022-887-3, 160 pp., 4³/₄ x 8

Using PageMaker 4 for Windows
Diane Burns & Sharyn Venit
Through Version 4
$29.95 USA
0-88022-645-5, 500 pp., 7³/₈ x 9¼

Quick & Dirty Harvard Graphics Presentations
Bob Benedict
Version 3
$24.95 USA
1-56529-089-5, 520 pp., 7³/₈ x 9¼

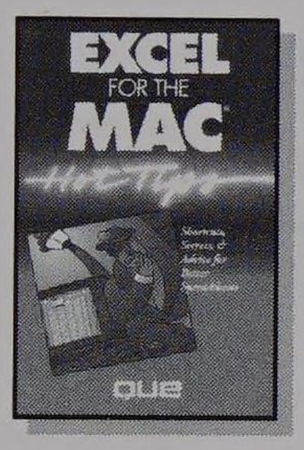
Using PowerPoint 3
Jim Meade
Through Windows Version 3
$29.95 USA
1-56529-102-6, 500 pp., 7³/₈ x 9¼

Using Ventura Publisher 4 for Windows
Skye Lininger
Versions 3 & 4
$29.95 USA
0-88022-929-2, 700 pp., 7³/₈ x 9¼

que **To Order, Call: (800) 428-5331 OR (317) 581-3500**

Improve with Hot Tips!

These unique guides teach readers shortcuts as well as powerful techniques—improving the proficiency of both novice and experienced users.

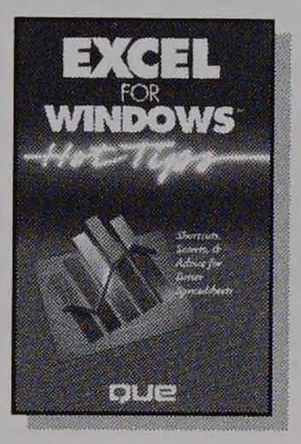


Excel for the Mac Hot Tips

Ron Person

Version 4

$12.99 USA
1-56529-162-X, 224pp., 5½ X 8½

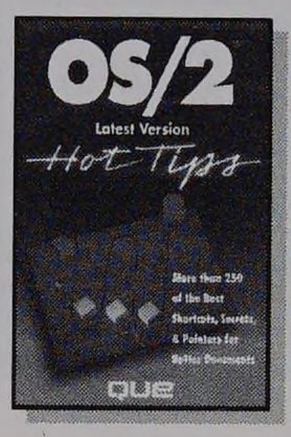


Excel for Windows Hot Tips

Ron Person

Version 4.0

$12.99 USA
1-56529-164-6, 224pp., 5½ X 8½

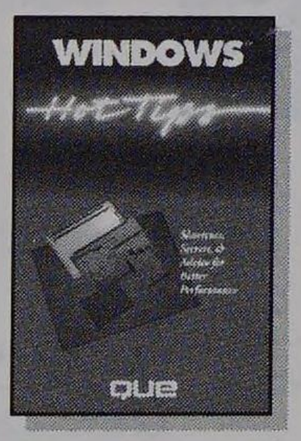


OS/2 2.1 Hot Tips

Que Development

Latest Version

$12.99 USA
1-56529-265-0, 224pp., 5½ X 8½

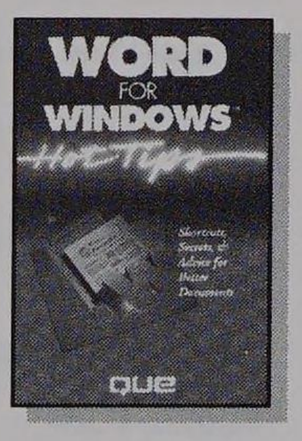


Windows Hot Tips

Steve Konicki

Version 3.1

$12.99 USA
1-56529-179-4, 224pp., 5½ X 8½



Word for Windows Hot Tips

George Beinhorn

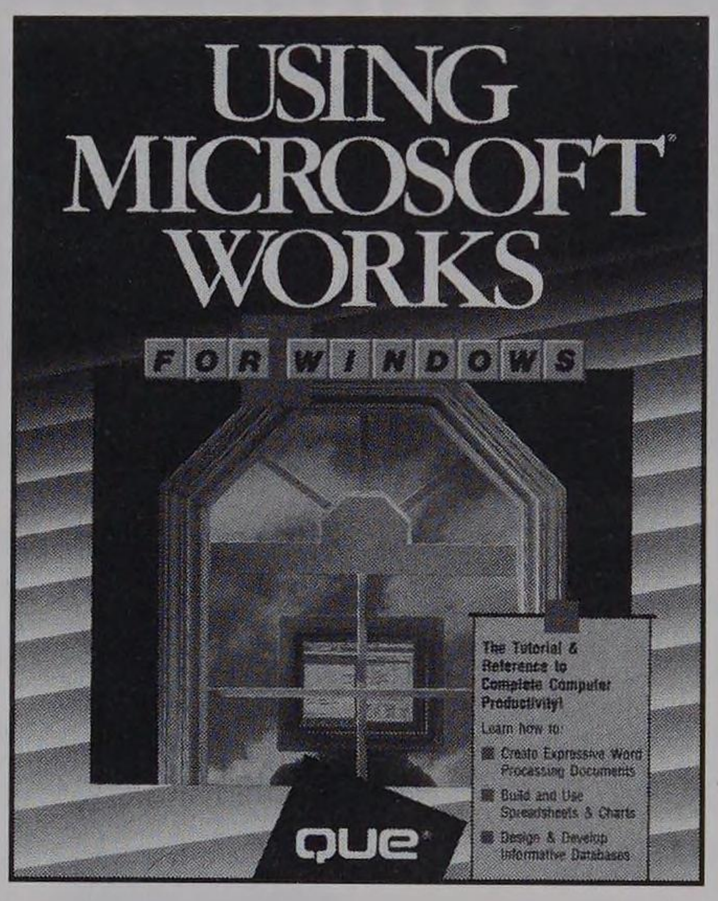
Latest Version

$12.99 USA
1-56529-163-8, 224pp., 5½ X 8½

que

To Order, Call: (800) 428-5331

Count on Que for the Most Up-to-Date Information on Integrated Packages

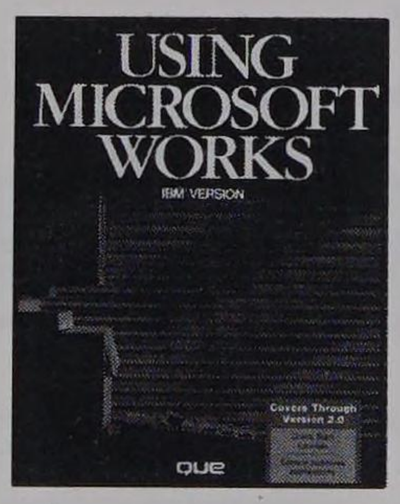


Using Microsoft Works for Windows, Special Edition
Douglas Wolf

This book provides plenty of timesaving tips and hints on sharing information between Works' modules. A comprehensive reference and tutorial, this book will give you everything you need to know about Microsoft Works.

Version 1

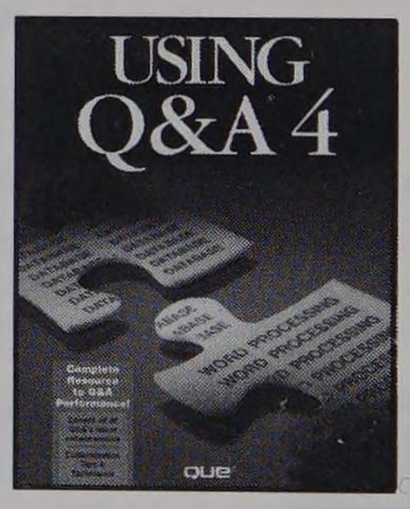
$24.95 USA
0-88022-757-5, 550 pp., 7 3/8 x 9 1/4



Using Microsoft Works: IBM Version
Douglas Wolf

Through Version 2

$22.95 USA
0-88022-467-3, 444 pp., 7 3/8 x 9 1/4



Using Q&A 4
David Ewing & Bill Langenes

Version 4

$27.95 USA
0-88022-643-9, 550 pp., 7 3/8 x 9 1/4

Q&A 4 Quick Reference
Que Development Group

Latest Version

$9.95 USA
0-88022-828-8, 160 pp., 4 3/4 x 8

Q&A 4 QuickStart
Que Development Group

Version 4

$19.95 USA
0-88022-653-6, 400 pp., 7 3/8 x 9 1/4

Que's Using Enable
Walter Bruce

All Versions Through Enable/OA4

$29.95 USA
0-88022-701-X, 700 pp., 7 3/8 x 9 1/4

Using Lotus Works 3.0
J. D. Watson

Release 3.0

$24.95 USA
0-88022-771-0, 500 pp., 7 3/8 x 9 1/4

Using PFS: First Choice 4.0
Katherine Murray

Through Version 4.0

$27.95 USA
0-88022-969-1, 600 pp., 7 3/8 x 9 1/4

Using PFS: WindowsWorks 2
Deanna Bebb

Through Version 2

$29.95 USA
1-56529-073-9, 500 pp., 7 3/8 x 9 1/4

que

To Order, Call: (800) 428-5331
OR (317) 581-3500

Polyray 1.6A Quick Reference

Controlling Polyray from the Command Line

The following table is a reference to all the command-line parameters available with POLYRAY. You can set many options of POLYRAY at the command line. The first column lists the parameter, and the second column describes the parameter's actions and uses.

NOTE: You can set up some of these parameters externally creating a file named POLYRAY.INI, the next section presents this method. Any parameter declared in the command line will override those inside the POLYRAY.INI file.

Example: POLYRAY demo.pi – o demo.tga –A –T 0.01 –S 15

where demo.pi is the input file, demo.tga is the output (image) file, use the adaptive antialiasing method with a threshold of 0.01 and the number of samples 15.

Command Line Parameters:

Parameter	Action
Antialiasing Parameters	
`-a`	Performs a simple antialiasing method of averaging the neighboring pixel colors.
`-A`	Performs an adaptive antialiasing method. It is dependent on the threshold value (parameter `-T`) and the number of samples per pixel (parameter `-S`).
`-J`	Performs a random (jittered) method.
`-T` *n*	Sets the threshold value to force over sampling. Over sampling is active if the difference in colors is greater than *n*. **Example:** `-T 0.1`
`-S` *n*	Sets the number of samples per pixel (*n*) for the adaptive (parameter `-A`) method. **Example:** `-S 6`
File Output Operations	
`-b` *n*	Saves the image data after calculating *n* pixels. This feature acts as a buffer to reduce the number of times that the hard disk is accessed. **Example:** `-b 1280`
`-B`	Saves the image data after calculating every line. Use this parameter if you are concerned with saving as much data as possible in case something goes wrong.
`-o` *f*	Specify the output file name *f*. If the parameter is not used, OUT.TGA is assumed. **Example:** `-o table.tga`
`-p` *n*	Set the number *n* of bits per pixel on the output image file. Acceptable values are 8, 16, 24, and 32. Default is 16. **Example:** `-p 24`
`-u`	Use uncompressed TARGA format on the output file.
Image Quality and Size	
`-r` *m*	Select method *m* to generate the image (the lower the number, the better the quality). **Value** **Method** 0 ray trace [default] 1 scan convert (requires substantial amounts of memory) 2 wireframe (to screen only) 3 Scan convert image and save as RAW data to a file. **Use as:** Polyray [parameters] –r 3 >filename.ext where filename.ext is the name of the output file. The RAW data file will consist of one line per triangle. The first nine numbers are the vertex coordinates. The next nine numbers are the components of the normal vectors of each vertex point in the corresponding triangle. Finally, the next six values are the local coordinates u1 v1, u2 v2, u3 v3 of each vertex. The coordinates u and v are the latitude and longitude coordinates of the triangle.

Image Quality and Size

`-d p` Dither objects with probability *p* (the number must be between 0 and 1). Use this parameter to reduce image generation time. Quality is also reduced.

Example: `-d 0.5`

`-D p` Dither *p* percent of rays used (the number must be between 0 and 1).

Use this parameter to reduce image generation time. Quality is also reduced.

`-q n` Select shading options:

Flag	Option
1	shadows
2	reflectivity
4	refraction
8	highlighting on both sides of a surface
16	check if object casts shadows
32	primary (eyes) rays will be checked

To use, simply add the number of the features wanted, for example,

```
shadows + reflectivity = 1 + 2 = 3.
```

The parameter usage would be `-q 3`.

Render Flag	Default
–r 0	– q 63 (all ON)
–r 1	– q 8
–r 2	N/A
–r 3	N/A

`-x` Set image x resolution, default 512.

Example: `-x 640`

`-y` Set image y resolution, default 512.

Example: `-y 480`

Miscellaneous

`-Q` Abort run by pressing any key.

`-R` Resume an interrupted trace.

`-t n` Display tracing information.

Value	Status
0	none
1	totals at the end of trace
2	line
3	pixel (slows program down!)

`-v` Ray trace image from bottom to top.

`-V m` Screen preview status.

Value	Mode
0	none
1	VGA 320 × 200
2	VESA 640 × 480 hicolor mode (use –t 0 also)

`-P n` Use palette number *n*.

Value	Colors
0	gray
1	666
2	884

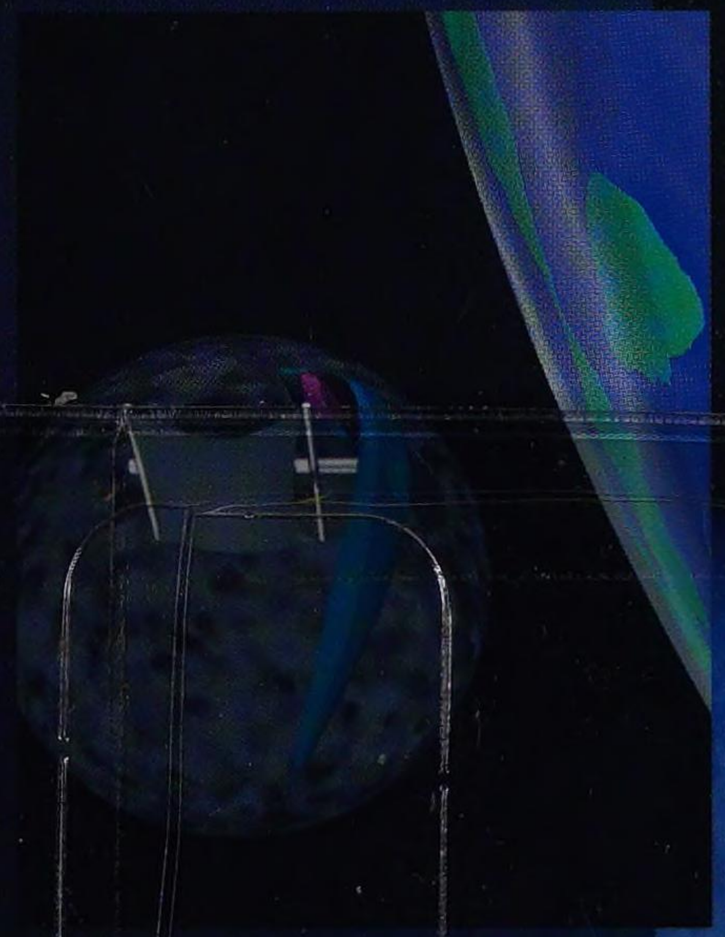
`-W` Wait for a keypress to clear the screen.

`-z n` Start ray tracing at line *n*. This feature is good for checking partial images.

POLYRAY.INI File

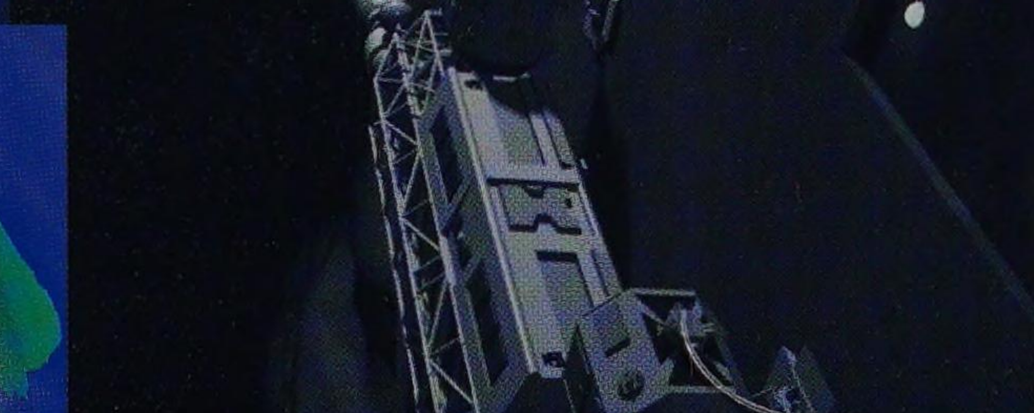
You may wish to set some of the options outside of the command line or the scene file. You can optionally create a file named POLYRAY.INI containing some or all of the following keywords followed by one of the corresponding values (this file must be accessible by Polyray):

Robert A. Mickelsen



Jerry Thompson

Dan Farmer



Heinz Schuller

Dan Farmer

Dan Farmer

Dan Farmer

Dan Farmer

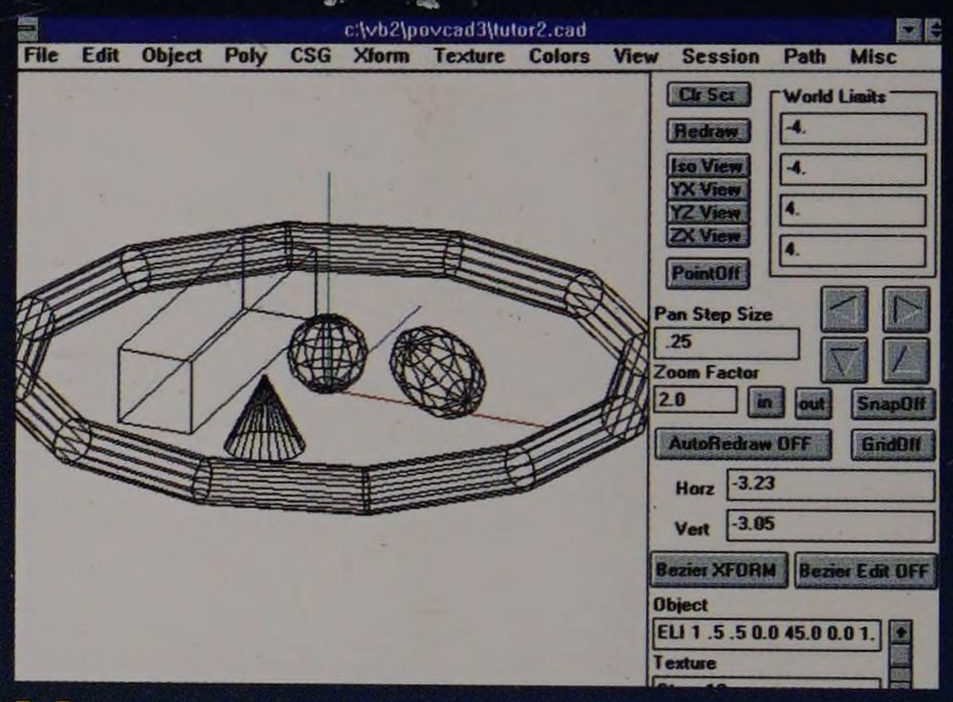
ADVENTURES IN RAY TRACING

625-3083
ADVENTURES
940-4C ADV COMP
18 01 02 3 1

A Guide to Dazzling Computer Graphics & Animation

- Generate your first ray traced image
- Invigorate your art with light, colors, shadows, reflections, and textures
- Build 3-D models on screen for constructive solid geometry
- Develop elaborate Polyray images for animation
- Create synthetic 3-D images with the POVCAD modeler
- Animate your ray-traced images for incomparable presentations

Welcome to the world of ray tracing, where the limitless fantasy of computer-generated art begins. *Adventures in Ray Tracing* covers the gamut—from wire frame plans to colorful drawings—to the ultimate quest-perfect photo-realistic images.



Use the included POVCAD program to generate the Polyray code for photorealistic images

Dan Farmer

Douglas Otwell

INFORMATION CENTER:
All User Levels
How To & Reference
IBM Compatibles
3D Graphics

P6-CAR-022

PRICE:
$27.95 USA
$34.95 CAN
£26.40 (Incl VAT) Net U.K.

ISBN 1-56529-555-2
90000
9 781565 295551